UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO DE FIN DE GRADO

ESTUDIO E INTEGRACIÓN DE UN SISTEMA DE CONTROL DE ENTORNOS MEDIANTE GESTOS USANDO KINECT

Jorge Lomas Rodríguez

SEPTIEMBRE 2013

ESTUDIO E INTEGRACIÓN DE UN SISTEMA DE CONTROL DE ENTORNOS MEDIANTE GESTOS USANDO KINECT

AUTOR: Jorge Lomas Rodríguez

TUTOR: Germán Montoro Manrique

Grupo de Herramientas Interactivas Avanzadas (GHIA) - Amilab

Ingeniería Informática

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Septiembre de 2013

RESUMEN

Este proyecto comienza con la necesidad de encontrar una manera sencilla y natural de comunicarse con un entorno inteligente. En el mundo de la inteligencia ambiental cada vez se pone más esfuerzo en que la interacción con el usuario sea lo más simple posible, ya que se desea que sea un elemento más de fondo, algo poco intrusivo, algo que no requiera aprendizaje. Por ello se decide hacer una interfaz de usuario natural usando el dispositivo Kinect.

Pero no todo queda ahí. Teniendo en cuenta las posibilidades del reconocimiento de gestos, se termina implementando un módulo capaz de hacerlo autónomamente y que pueda ser reutilizado para cualquier otro tipo de proyectos.

Partiendo de un entorno inteligente creado anteriormente, AMILAB, este trabajo de fin de grado muestra una manera sencilla pero potente de llevar a cabo dicha solución utilizando el reconocimiento de esqueleto que devuelve la Kinect y evaluando el punto del mismo que se le asigna a la mano derecha. Esta será la encargada de realizar los gestos.

Palabras clave:

Gesto, movimiento, reconocimiento, entornos inteligentes, inteligencia ambiental, interfaz de usuario natural, interfaz de usuario gráfica, librería, control, Kinect, esqueleto, sensor de profundidad, cámara.

ABSTRACT

This project begins with the need of finding a simple and natural way to communicate with an intelligent environment. In the world of ambient intelligence it's beginning to put more effort on making interaction user-machine simpler because it's needed to be a background, little intrusive and easy to learn item. That's why I decided to make a natural user interface using Kinect.

But that's not all. I finally made a module that could recognize gestures autonomously and be reused for other projects, a library.

That end of degree project shows a simple and potent way to implement that solution, starting with the intelligent environment made before in AMILAB, using the skeleton tracking provided by the Kinect and evaluating the right hand points that it gives us. That hand will have the charge to do the gestures.

Key words:

Gesture, movement, recognition, intelligent environment, ambient intelligence, natural user interface, graphical user interface, library, control, Kinect, skeleton, depth sensor, camera.

AGRADECIMIENTOS

Realmente, gracias a todos. Gracias por enseñarme a vivir. Gracias por enseñarme a luchar. Gracias por enseñarme a no rendirme y, si me caigo, levantarme. Gracias por vuestro tiempo, cariño, apoyo y, también al contrario, gracias a los que no lo dan por enseñarme a distinguir quién estará ahí, quién me ayudará, quién me guiará y quién, sin embargo, solo estará de paso.

A un paso de terminar. A un paso de cerrar esta etapa de mi vida. Salgo contento y triste, con muchas ganas y a la vez mucho miedo. Salgo sabiendo que no es oro todo lo que reluce, y muchas cosas buenas se quedarán atrás. Pero al menos espero que siga contando con la gente que me quiere y yo quiero. Mi gente es la gasolina que hace girar el mundo y espero que los que han entrado en esta etapa, sigan girándolo durante mucho tiempo.

Gracias a la universidad y todos los que están (o estaban) en ella. Ha sido una etapa larga y bastante dura, pero los momentos que uno se lleva no son esos. Gracias a mis compañeros por esas risas, por enseñarme a dejar a un lado el trabajo de vez en cuando, despejarme y seguir con más fuerza. Gracias, además, a todos aquellos que se han ido, o se irán. Me habéis enseñado a trabajar en equipo, a conseguir lo que quiero y, en algunos casos, aunque sea triste, que hay que separar el trabajo de la amistad. Gracias a mis profesores, a muchos de ellos. He aprendido mucho con sus palabras, y no sólo de informática. Espero ser lo que esperáis de un alumno vuestro. Gracias a mis lugares de trabajo y su gente. Me habría aburrido mucho sin ellos. La verdad, no concibo esta etapa sin ciertos momentos. Me han enseñado a trabajar duro, a hacer cosas aburridas, difíciles, imposibles, y sentirme reconfortado cuando se ve el final. Y gracias al AMILAB y los que están en él. La vida se ve diferente cuando estás ahí. Gracias por encauzarme a ser un buen profesional (y por aguantarme todo el año, verano incluido).

Y sobre todo, gracias a mi familia y amigos. Mis padres, mi hermano, mis tíos, mis abuelos, mis primos, los frikis de Madrid, los de Colme, los del curro y los de siempre. Gracias por aguantarme, se que he sido un poco insoportable por momentos. Gracias por entenderme y saber cuando necesito mi tiempo. Gracias por no dejarme tirado en momentos de necesidad, aunque mis palabras os forzaran a ello. Gracias por estos años, y los que vienen.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1 INTRODUCCIÓN

Los **entornos inteligentes**, últimamente, se están convirtiendo en el día a día del usuario medio. Los televisores, neveras y diferentes electrodomésticos cada vez están más conectados y ofrecen un mayor número de posibilidades. Entornos programados para, en caso de, por ejemplo, vacaciones, pongan diferentes servicios en funcionamiento, electrodomésticos que ayuden en el día a día a personas con algún tipo de discapacidad (ya sea notificándoles, guiándoles, automatizando acciones, …).

Poco a poco las nuevas tecnologías van avanzando y con ello se lleva a cabo la creación de elementos más eficientes para llevar a cabo todas esas tareas: tablets, teléfonos inteligentes (o *smartphones*), controladores (ratones, teclados, mandos), etc. Con ellos el intercambio de información entre dispositivos del entorno es cada vez más rápido y eficiente, mejorando mucho el tiempo de uso y de aprendizaje, además de gestionando de una manera más eficiente e intuitiva todos los recursos que ofrece cada dispositivo. Gracias a ello se está haciendo mucho énfasis en nuevas investigaciones como las llamadas **interfaces naturales de usuario** (o ***NUI*** por sus siglas en inglés).

Dispositivos como **Kinect**, Leap Motion y Wiimote, entre otros, están abriendo una gran variedad de puertas a la hora de crear NUIs e investigar en ello. Campos tan diversos como la medicina (ayudando en quirófanos, por ejemplo, a tener el menor número de contactos con agentes externos y evitar problemas en una operación) o los videojuegos (con juegos como los de baile) están fomentando la aparición de estos aparatos y la investigación, como el que nos lleva a continuación.

1.1 Motivación

Este proyecto nace con la idea de, utilizando un sistema ontológico creado en el laboratorio AMILAB, la pizarra (la cual discutiremos en detalle más adelante), que describe un entorno inteligente, comunicar algún dispositivo que reconozca los movimientos llevados a cabo por un usuario para controlar los elementos de cualquier entorno mediante gestos complejos (tales como un cuadrado).

Además, un gran número de investigaciones ronda en torno al uso de NUIs que permitan a diferentes tipos de usuarios (incluidas personas con diferentes tipos de discapacidades) utilizar elementos, que de otra manera les sería imposible, además de ayudar muchas veces en rehabilitaciones y mantenimientos de los mismos.

La **Kinect de Windows** se ajusta en gran medida a ello con lo cual se utilizará para llevar a cabo el proyecto, aunque, como se comentará más adelante, se podría llevar a cabo con cualquiera e, incluso, crear una biblioteca independiente en la que solamente se tuviera que indicar el dispositivo con el que se captarán los movimientos.

1.2 Objetivos

Durante el presente **trabajo de fin de grado** (o **TFG**), se tratará de, mientras se investiga la mejor manera de llevar a cabo un reconocimiento de gestos con el Kinect, desarrollar una biblioteca que, independiente del sistema que la llame o use, pueda hacer uso del dispositivo de Microsoft para reconocer un número de gestos determinado por el desarrollador. Para ello, se tendrán en cuenta:

- La descripción de los gestos ha de ser fácil de entender e intuitiva a la hora de crear nuevos.
- Los gestos se dividirán en movimientos para que a su vez, en caso de que un usuario desee utilizar un mayor número o menor número de gestos, pueda ajustarlos a los movimientos deseados.
- La biblioteca tendrá que pedir un coste bajo al procesador. Puede que el programa que haga uso de la misma se encuentre en la misma máquina y se necesite un rendimiento alto para obtener la mayor información posible en el menor tiempo.

- No deberá suponer un coste de aprendizaje alto el uso de la misma. Se abstraerá al usuario de toda la gestión posible.
- Ha de ser altamente modular y modificable. Según pase el tiempo tendrá que poder adaptarse a nuevos dispositivos y nuevas investigaciones que surjan en el campo, por lo tanto se necesita algo tan modular que se pueda actualizar sin necesitar de grandes cambios en el grueso de la biblioteca.

Además, se desarrollará una interfaz con el entorno inteligente (lo cual es realmente el tema del proyecto) que, haciendo uso de la biblioteca que desarrollaremos, comunicará el entorno inteligente del AMILAB con el Kinect.

- Ha de ser intuitiva y rápida. No se necesita gran información para su uso, no se ha de saturar la **GUI** con indicaciones irrelevantes.
- Tendrá que indicar en todo momento qué movimientos están disponibles.
- Cualquier tipo de usuario deberá poder usarla, independientemente de los conocimientos del usuario en informática.

1.3 Organización de la memoria

Durante esta memoria describiremos el proyecto llevado a cabo durante todas sus fases. Se tratará de explicar, con el mayor detalle posible, qué se ha hecho, por qué se ha decidido hacer y cómo hacer, cómo se ha hecho y, finalmente, si se ha conseguido un funcionamiento óptimo y, en caso de no ser así cómo mejorarlo.

En primer lugar, en el capítulo dos, hablaremos sobre el **estado del arte**. Aquí discutiremos todo lo que ronda alrededor de nuestra aplicación (las diferentes tecnologías que se usan y sus variantes, por qué se ha comenzado a llevar a cabo este tipo de investigaciones, cómo se ha tratado el tema en otras investigaciones, …).

En el tercer capítulo trataremos el **análisis y diseño** del proyecto. Describiremos el problema que nos trae a la investigación, que herramientas (tanto *hardware* como *software*) serán usadas y por qué y el diseño de nuestro proyecto.

Más adelante, en el capitulo cuatro, se hablará sobre el **desarrollo** tanto de la librería como de la interfaz y comunicación con el entorno inteligente. Como se describirá a lo largo de esta memoria, se ha tenido que cambiar radicalmente el punto de vista del desarrollo, con lo que se ha necesitado reimplementarlo todo, se hablará tanto del primer desarrollo como del segundo. Además, se detallarán los problemas encontrados y las soluciones propuestas a los mismos (si se han llevado a cabo, el resultado final).

Y finalmente, pero no menos importante, comentaremos cómo se puede ampliar el proyecto, hacia qué ramas se podría dirigir y cómo se podrían mejorar los diferentes módulos del mismo, además de aportar una conclusión, en el quinto y último capítulo, **conclusiones y trabajo futuro**.

CAPÍTULO 2 ESTADO DEL ARTE

La Inteligencia Ambiental (*Ambient Intelligence* o *AI*) provee «entornos electrónicos adaptativos que responden a las acciones de personas y objetos mientras satisfacen sus necesidades» [1]. Con el paso de los años se han incrementado el número de entornos inteligentes además de las soluciones que ellos proveen.

En el campo de los Entornos Inteligentes (*Intelligent Environments*) se estudia «cómo el uso de ayudas tecnológicas mejora la experiencia humana en el trabajo, la vida diaria, transporte y otros. Estas mejoras pueden ser tanto en productividad como en confort o interacciones sociales» [2]. Para todo ello, se han desarrollado con el paso de los años herramientas cada vez más capaces, cada vez menos intrusivas y cada vez más fáciles de usar. Pero, ¿qué es una buena herramienta? [3]. Una herramienta invisible, a ser posible. Que el ser humano no note su presencia, no precise de demasiado aprendizaje y sea capaz de aportar toda la información necesaria al entorno para su funcionamiento.

Según avanza la tecnología van apareciendo mejores herramientas que aportan al usuario una manera natural de comunicación para con la máquina, una Interfaz de Usuario Natural (*Natural User Interface* o *NUI*). Estas, como su nombre indica, serán naturales al usuario, no necesitarán de demasiado entrenamiento previo y el usuario disfrutará usándolas [4]. Por ello, la gran mayoría de estas herramientas o son creadas para el ocio (en videojuegos, centros multimedia, …) o acaban usándose en ese ámbito. De las más importantes son las cámaras que han desarrollado tres de las más grandes compañías de videoconsolas: Nintendo (WiiMote) y Microsoft (Kinect). Aunque orientadas a los videojuegos, se han comenzado a usar en un gran número de proyectos en muchos y variados ámbitos que demuestran el potencial de las mismas y todos los usos que se les pueden dar (v. gr. en el tema que nos concierne, el reconocimiento de gestos y movimientos para controlar entornos inteligentes).

2.1 Interfaces de usuario naturales (NUI)

Evolucionadas desde las simples CLI (*Command-Line Interface*), primeras interfaces de usuario para con la máquina que usaban lo escrito por el usuario para comunicarse con la máquina, y las GUI (*Graphical User Interface*), las cuales ya incluyeron indicadores gráficos tales como botones y otros iconos (que proporcionaron una mejora muy grande con respecto a la curva de aprendizaje de las CLI), las NUI (*Natural User Interface*) nacen en 2006 de mano de Christian Moore, el cual creó la NUI Group Community para expandir el desarrollo de las mismas [5].

Las NUI proponen crear una interfaz de usuario natural, con un coste de aprendizaje mínimo. En el proyecto de Evers-Senne et al. [6] se puede ver como se puede desarrollar una NUI que reconozca la posición del usuario y diferentes movimientos del cuerpo para navegar a través de una escena sin necesidad de usar ratones u otros periféricos. Brandl et al. [7] también proponen una diferente interfaz natural con su sistema de, a través de la escritura natural de un usuario en un cuaderno, almacenar notas y demás información de interés.

Nosotros nos centraremos en las NUI que se pueden crear gracias a, como comentaba anteriormente, las cámaras desarrolladas por las compañías de videojuegos.

2.1.1 WiiMote

Nacido en Noviembre de 2006, el Wii Remote (o *WiiMote*) ha sido un éxito global en el mercado. Tanto para el sector de los videojuegos (el WiiMote ofrece una nueva y más intuitiva manera de interaccionar con los videojuegos incluyendo, además de los botones de siempre, reconocimiento de gestos y localización donde apunta el usuario) como para la investigación (numerosos proyectos se han aprovechado de sus capacidades [8]).

Desde estudios que demuestran cómo con el uso del WiiMote ayuda a niños con déficit de atención e hiperactividad a mejorar su auto-control gracias al uso del mando (como en el llevado a cabo por Shih et al. [9]) hasta sistemas para ayudar en el aprendizaje en el campo de la medicina donde poder practicar de una manera más directa (W. John [10]).

2.1.2 Kinect

Otra de las más importantes cámaras desarrolladas en los últimos años ha sido la Kinect de Windows. Su mejor baza frente a otras es el hecho de no necesitar de un hardware externo para transmitir información, las NUI desarrolladas con ella pueden ser movimientos realizados en el aire y por cualquiera, no necesitas buscar un mando para comunicarte.

Se han ido llevando a cabo multitud de proyectos con la misma. Oikonomidis et al. [11] han desarrollado, por ejemplo, un sistema para reconocer las articulaciones de la mano usando Kinect. Joel Filipe [12] propone un sistema de aparcado autónomo usando la cámara de profundidad de la Kinect. También se ha llevado a la medicina y se han estudiado proyectos tales como el de Chang et al. [13] que propone un sistema de rehabilitación para adolescentes con discapacidades motoras.

2.2 Reconocimiento de gestos, movimientos, figuras...

Una de las maneras de comunicarse con las máquinas implementando NUIs más investigadas de los últimos años es a través de cámaras. Hacer movimientos, gestos, figuras, etc., es la forma más útil e intuitiva que se está desarrollando. Mistry et al. [14] han desarrollado un sistema que, usando una cámara, un proyector de bolsillo y un espejo, lleva una interfaz a cualquier parte y captura los gestos de la mano. Dave et al. [15], por el contrario, han desarrollado un sistema de interacción con un ordenador a través de gestos (teniendo en cuenta los típicos que se asocian a un ordenador, tales como el movimiento del ratón y el doble clic).

Además, se ha ido avanzando mucho en el reconocimiento de formas para su uso en NUIs. Un buen ejemplo de como reconocer diferentes formas (aunque no sean cuerpos humanos) sería el de Belongie et al. [16]. Otro grupo, Plagemann et al. [17], con el uso de una cámara que devuelve imágenes en profundidad es capaz de encontrar y descomponer el cuerpo humano para su uso en diferentes aplicaciones, o el de Shotton et al. [18], el cual consigue un mejor reconocimiento ya que divide el cuerpo en treinta y una partes diferentes (frente a las cinco reconocidas en el anterior).

Gracias a ellos, también se comienza a investigar en el reconocimiento real de gestos y movimientos. Ganapathi et al. [19], Bobick et al [20], Chen et al. [21] y Li et al. [22] Han desarrollado lo mismo de diferentes maneras. Desde el análisis de un montón de puntos a analizar, usando unas plantillas creadas para la ocasión o un conjunto de imágenes con profundidad.

Finalizando, con el uso de los dispositivos atrás mencionados (WiiMote y Kinect), también se está indagando en el campo del reconocimiento de gestos. Las funcionalidades de estos aparatos facilitan al investigador en su tarea, puede abstraerse de muchos cálculos para poder dedicarse al cien por cien al fin. Un ejemplo de ello es de Schlömer et al. [23] que, gracias al uso del WiiMote, desarrollan un sistema de reconocimiento de gestos con respecto a una pequeña muestra predefinida. Además, consiguen un muy buen sistema de filtrado para los mismos. En cuanto al Kinect, casi todo lo que se ha desarrollado acaba siendo un sistema de reconocimiento de gestos de una mano, hacer un símbolo con la misma y que el sistema lo reconozca (Ren et al. [24]).

2.3 Conclusiones

Aunque está bastante poblado el tema de reconocimiento de objetos a partir de imágenes, el hecho de reconocer gesto es un buen nicho de investigación. No solo porque no hay demasiados estudios respecto al tema (aunque los haya, pero son más orientados a formar una imagen en el tiempo y evaluarla al final) si no porque los que realmente hacen algo parecido son muy restringidos. No son flexibles. Crean un banco de gestos y solo se pueden usar los mismos. En este proyecto se tratará de hacer algo flexible, fácil de usar y que se pueda usar en una gran variedad de entornos, crear una NUI que se pueda reutilizar.

CAPÍTULO 3 ANÁLISIS Y DISEÑO

3.1 Descripción del problema

Al tratar sobre temas de espacios inteligentes siempre surgen ciertos problemas: ¿cómo se podría comunicar el ser humano de una manera más eficaz con el entorno? De ahí surge el siguiente proyecto. Tras comprobar que casi todas las interfaces con la máquina acaban necesitando de periféricos intermedios, se quiere tratar de evitarlos. Se necesita un medio en que cualquier persona pueda utilizarlo sin casi conocimientos previos. Fácil de usar, intuitivo, adaptable, …, son varias de las necesidades en cuanto a un desarrollo orientado a cualquier tipo de usuario. Por ello, se plantea el uso de nuevas tecnologías como *smartphones* o cámaras para el uso de los mismos.

A continuación, se plantearán las opciones tenidas en cuenta, sus ventajas y desventajas, las necesidades que se desean cubrir y cómo, una vez elegidas las tecnologías, se desarrollará el proyecto (arquitectura e interfaz).

3.2 Consideraciones en el diseño

Previo al desarrollo, una de las cuestiones importantes que se tuvo que tener en cuenta fue el hecho de qué hardware, software y diferentes tecnologías elegir. En los siguientes apartados se tratará de:

- Dar a conocer las diferentes tecnologías usadas.
- Tratar el por qué se han elegido.
- Mostrar las diferencias (beneficios y desventajas) con respecto a otras tecnologías disponibles en el mercado.

3.2.1 Kinect

El **Kinect de Microsoft [25]** es, básicamente, una cámara, pero una cámara con multitud de funcionalidades incluidas a las que, gracias al **API** aportado por Microsoft, podremos acceder con facilidad y obtener buenos resultados sin la necesidad de implementar las mismas.

Ilustración 3.1*: Kinect, la cámara de Microsoft.*

Kinect es una cámara que fue desarrollada orientada a los videojuegos a finales de 2010. Su función fue, realmente, la de permitir a los jugadores interaccionar con la consola y sus videojuegos favoritos obviando la necesidad de un periférico intermedio, eliminando el mando que, hasta entonces, pocas veces se había podido eliminar de esta afición con resultados satisfactorios. Antes de ella, otras compañías intentaron lo mismo pero sin grandes resultados. Sony, con EyeToy estuvo cerca, pero al final quedó relegado su uso a videojuegos más enfocados a un público casual que al jugador más constante.

Esta cámara contiene, además de las funcionalidades de una cámara normal (ver y captar imágenes a color), varias funcionalidades incluidas. Puede desde captar sonido y tratarlo hasta, gracias a unos sensores de profundidad, devolver el esqueleto de una persona para su uso y análisis.

Como se puede comprobar el la **ilustración 3-2**, Kinect cuenta con variedad de hardware para facilitar su uso y recogida de datos [26]:

- Sensores de profundidad en 3D (*3D Depth Sensors*) con los cuales se puede obtener una imagen en profundidad además de las cordenadas X, Y y Z de todos los puntos de la imagen. También tiene disponible la opción de reconocer y devolver cuerpos marcando los puntos críticos de los mismos.

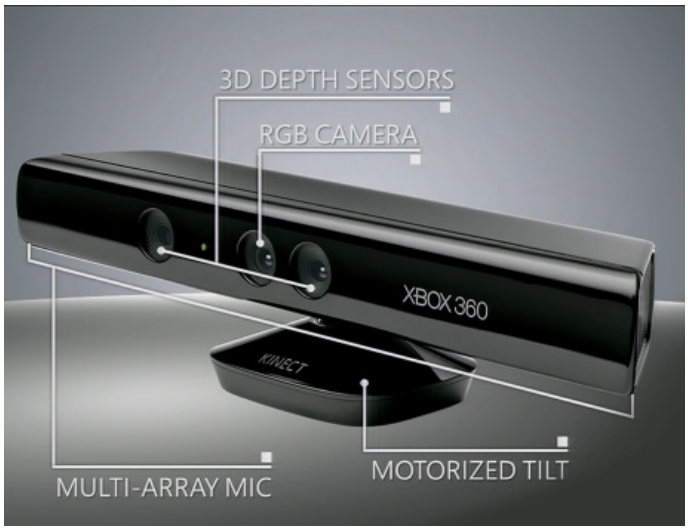


Ilustración 3.2*: Partes de la Kinect.*

- Cámara a color (*RGB Camera*) que nos devolverá la imagen real que está viendo la misma. Uniéndola con los sensores de profundidad se obtiene mucha información del entorno frente a la Kinect.
- Fila de micrófonos (*Multi-Array Mic*) la cual ejercerá de micrófono a la antigua usanza además de facilitar el reconocimiento de voz y situación de la misma.
- Inclinador motorizado (*Motorized Tilt*) al cual se puede acceder a través del mismo programa y cambiar la inclinación de la cámara.

Todo ello hace a la Kinect una gran cámara, no solo orientada a los videojuegos, si no también da posibilidad real a multitud de proyectos basados en ella gracias a un precio muy competitivo viendo la multitud de posibilidades de la misma que, en caso de llevarlo a cabo con otros instrumentos, puede salir muy caro en tiempo y dinero.

Aún con todo lo anterior, hay una gran variedad de tecnologías en la actualidad que podrían haber servido para implementar el proyecto, no solo se puede tratar de controlar el entorno a través de gestos con el Kinect. Las nuevas tecnologías móviles (tales como *tablets* y *smartphones*) posibilitan su uso desde cualquier posición con unos simples gestos en el móvil o un par de clics.

Además, las demás grandes compañías de videojuegos también están tratando de coger un trozo del pastel con sus propias soluciones. **Nintendo**, con su **Wii** ha captado gran parte de ese mercado gracias al uso de un mando, el **WiiMote** [27] con acelerómetro y una barra de sensores con leds infrarrojos que ayudan al mando a indicar la posición. **Sony** también ha indagado en este terreno y ha creado el **PlayStation Move** [28] que, gracias al uso de su **PlayStation Eye** [29] , un acelerómetro y una esfera que se ilumina en la parte superior del mando, también es capaz de captar los movimientos y posición de los mismos.

Ilustración 3.3*: WiiMote y PlayStation Move.*

Lo malo de todas las soluciones propuestas es la necesidad de un elemento intermedio que transmita los datos a la consola. Con este proyecto se desea interaccionar con cualquier sistema de una manera directa e intuitiva y, aunque para la gran mayoría del público el uso de un mando es algo de su vida cotidiana, estos elementos añaden un aprendizaje intermedio además de incluir el inconveniente de necesitar ese hardware, no puedes llegar y actuar, se ha de encontrar, sacar del bolsillo, … antes de poder comenzar con su uso.

Una vez decidido el uso del Kinect como hardware intermedio para la gestión del entorno, se planteó una nueva duda: Microsoft ha desarrollado dos Kinects diferentes. La primera que salió al mercado fue **Kinect for Xbox** que, aunque llevaba consigo todas las funcionalidades, no era tan completa como **Kinect for Windows** la cual salió a la venta casi un año después. A continuación, en la **Tabla 3-1** se detallan las diferencias entre ambos modelos.

	Kinect for Xbox	**Kinect for Windows**
Plataformas	Xbox y PC (Windows) mediante un cable USB.	Windows 7 o Windows 8.
Rango de profundidad	1,2 a 3,5 metros.	Desde los 0,4 a 4 metros, aunque puede llegar a los 8 con una información no óptima.
Modos de profundidad	320 x 240. 30 fps.	80 x 60, 320 x 240 o 640 x 480. 30 fps.
Modos de color	640 x 480. 30 fps.	640 x 480. 30 fps. 1280 x 960. 12 fps. Raw YUV 640 x 480. 15 fps.
Audio	16 bits.	32 o 64 bits.
Sensores simultáneos por equipo	1.	4.

Tabla 3-1*: Diferencias entre Kinect for Xbox y Kinect for Windows.*

Viendo las características anteriores y teniendo en cuenta que el laboratorio ya disponía de una Kinect for Xbox se decidió usar la misma para el trabajo. Las novedades con respecto a la Kinect for Windows no son destacables para nuestras necesidades (no usaremos más de un dispositivo, no necesitamos un rango excesivamente grande ni pequeño de momento, …). A partir de este punto cuando se hable de Kinect será en referencia a Kinect for Xbox.

3.2.2 Microsoft Visual Studio 2010

Es un **IDE** (entorno de desarrollo integrado, o *Integrated Development Environment* en ingles) para desarrollar en sistemas operativos **Windows** [30]. Nació a mediados de 1998 y actualmente se encuentra en su versión 11 (**MS VS 2012**). Permite desarrollar en multitud de lenguajes, pero sobre todo se basa en lenguajes propios de Microsoft tales como ASP.NET, Visual Basic, C#, …

Se ha decidido desarrollar en el mismo ya que el SDK de la Kinect que ofrece Microsoft está disponible para C# y Visual Basic. También se podrían elegir SDKs de código abierto pero, a sabiendas de que estos pueden llevar a cabo funcionalidades alternativas, son desarrollados sobre todo por ingeniería inversa, con lo cual, para ciertos procesos, no serán igual de eficientes. Además, nuestras necesidades las cubre perfectamente el SDK oficial e incluye una gran documentación online que es de gran ayuda.

3.2.2.1 C#

Lenguaje de programación orientado a objetos (**OOP**) propietario de Microsoft. Nacido hacia el año 2000, permite crear y ejecutar diversas aplicaciones corriendo sobre **.NET Framework**. Actualmente la versión de C# es la 4.0.

Durante su definición, se trató de atraer al mayor número de desarrolladores. Para ello, han construido su sintaxis de una manera similar a Java, C o C++, los cuales eran los lenguajes de programación más usados en la época. Está basada en símbolos de llave, permite asignar valor de *NULL* a cualquier objeto, permite definir métodos y tipos genéricos, utiliza las expresiones **LINQ** para crear consultas a bases de datos con una estructura de lenguaje de alto nivel, …

Además, como lenguaje orientado a objetos, admite la encapsulación, herencia y polimorfismo. Al igual que Java, no permite herencia múltiple pero si implementar cualquier número de interfaces. Las novedades frente a otros lenguajes serían:

- Firmas de métodos encapsulados.
- Propiedades (descriptores de acceso de variables privadas).
- Atributos (metadatos en tiempo de ejecución).
- Genera documentación en XML a partir de comentarios.
- Introduce **LINQ** que ayuda a abstraer el origen de datos de las consultas.
- Gran interoperabilidad con los diferentes recursos de Windows.

Se ha decidido por este lenguaje de programación ya que, como hemos comentado anteriormente, Microsoft sólo ofrece SDK para Kinect en C# o Visual Basic. Además, C# es un lenguaje en pleno auge y servirá como experiencia laboral real. En cualquier caso, la experiencia previa en prácticas en empresa y en alguna asignatura de la carrera hacen que C# sea mi opción, ayudado por el hecho de que nunca he desarrollado en Visual Basic y podría llegar a salir de los límites del TFG el aprender otro nuevo lenguaje de cero.

3.2.2.2 Windows Presentation Foundation (WPF)

Marco de trabajo para realizar aplicaciones cliente enriquecidas e interactivas. Se preparó para que el desarrollo con el mismo fuera una experiencia familiar para cualquier desarrollador que hubiera programado anteriormente en ASP.NET o Windows Forms, utilizando un lenguaje XAML para ello.

Su núcleo es un motor de representación basado en vectores e independiente de la resolución de la pantalla que trata de hacer uso de las prestaciones de las *GPU* modernas. Se incluye dentro del .NET Framework de Microsoft de modo que puede incluir otros elementos de las bibliotecas de clases correspondientes.

Utiliza un lenguaje de marcado, **XAML**, basado en **XML**, por lo que los objetos se ensamblan en una jerarquía de elementos anidados llamada **árbol de elementos**.

Para controlar la interacción con el usuario, WPF incluye el llamado *code-behind* que se ocupará de la lógica de la página (llamadas a eventos, clics del usuario, acceso a datos, …). Este código será desarrollado en C#, aunque hay varios lenguajes posibles, ya que toda la lógica de la librería la hemos desarrollado en el mismo.

Se ha decidido usar este marco de trabajo para adaptarse a las tecnologías del momento. Windows Forms, por ejemplo, se está quedando anticuada y no saca el partido que WPF puede sacar de un sistema Windows. Además, es muy intuitivo y generar la interacción de la ventana con el código desarrollado es ampliamente sencilla. Al tener conocimientos previos con ASP y ASP.NET la dificultad no será excesiva y no se tendrá que extraer demasiado tiempo del fijado para el TFG, el cual se podrá tener en cuenta para el desarrollo general. Además, al igual que para C#, Microsoft ofrece una amplia documentación, organizada e intuitiva, que ayuda al desarrollo.

3.2.3 NetBeans IDE

Al igual que MS VS, NetBeans [31] es un IDE. Está orientado principalmente a la programación en lenguaje Java, pero soporta la gran mayoría de lenguajes actuales. Es libre, gratuito y, además, tiene una gran comunidad de desarrolladores detrás. Al ser de código abierto y libre, tiene una gran variedad de módulos que extienden su funcionalidad. Se encuentra actualmente en la versión 7.3.

La mayor diferencia de Netbeans con respecto a otros IDEs como Eclipse [32] es su facilidad de uso. Aunque tiene un menor número de *plugins* o extensiones también hace que los plugins que existen sean más consistentes en el primero.

Finalmente se va a optar por desarrollar en Netbeans ya que se tiene experiencia previa en el uso de la herramienta y no conlleva a un aprendizaje.

3.2.3.1 Java

Lenguaje de programación orientada a objetos desarrollado por **Sun Microsystems** (ahora propiedad de **Oracle**) por el año 1995. Fue desarrollado pensando en la facilidad de los desarrolladores de programar de una vez el programa y que fuera posible su uso en cualquier sistema operativo. Para ello, hace uso de una máquina virtual (**JVM**, **Java Virtual Machine**), la cual si que tiene un desarrollo especial para cada sistema operativo y versión del mismo, pero que hace que, al lanzar un programa en la misma, haya una abstracción del entorno en el que corre esta. Además, se desarrolló el recolector de basuras (o *automatic garbage collector*) que hace que el programador olvide la gestión de la memoria: una vez el objeto deja de usarse, este pasa a ser controlado por el recolector (controlado por el **Java Runtime**) que, durante la ejecución, cuando cree que este no va a ser necesitado nunca más, lo elimina y libera su memoria.

Como se comentaba, Java ha sido creado para ejecutar en multitud de entornos tales como: dispositivos móviles, sistemas empotrados, navegadores web, sistemas de servidor y programas de escritorio.

Uno de los principales problemas que se encuentran en Java es el consumo de recursos y tiempo de ejecución. Al necesitar del uso de una máquina virtual, el consumo suele ser mayor que si lo comparamos con otros lenguajes. Además, el recolector de basura, por ejemplo, aunque beneficioso, añade una pequeña sobrecarga al proceso.

La pizarra (el módulo para comunicarse con el entorno que se presenta a continuación), ha sido desarrollada en Java y tiene casi todo ya desarrollado para ese lenguaje. Por ello, se ha tenido que elegir esta opción para llevar a cabo el servidor que escuchará al programa.

3.2.4 Pizarra (*Blackboard*)

La Pizarra es el sistema usado en el laboratorio AMILAB [33] para el control del entorno. Es una capa intermedia entre los elementos del entorno y las aplicaciones que los gestionan y reciben información de los mismos. La metáfora de la pizarra implica que todo intercambio de información se encuentra centralizado en un mismo punto donde los productores de información la publican en ella sin saber quién podrá consumirla.

El modelo de datos utilizado representa toda la información relativa al entorno, independientemente de la fuente de la misma. Está dividida en dos partes muy diferenciadas:

- El esquema, el cual contiene la descripción del entorno en clases, sus propiedades, competencias y las relaciones que mantienen entre ellos, lo cual es un modelo ontológico.
- El repositorio, el cual mantiene las entidades que implementan esas clases definidas en el esquema. Pueden representar objetos tanto reales como virtuales.

Además, sirve como una fuente de información en la que las aplicaciones conectadas acceden a ella sin saber de donde procede, no tienen conocimientos extra del resto del sistema, solo de la pizarra. Esto abstrae de la comunicación entre diferentes procesos lo que consigue que se haga más fácil el desarrollo de tareas sin perder la escalabilidad del sistema.

Para la comunicación en el sistema, se establece una división de los diferentes módulos, entre reales o virtuales (**Ilustración 3.4**). Se tienen cinco tipos de clientes: sensores, actuadores, intérpretes, consumidores y realizadores. La pizarra, como se ha comunicado anteriormente, hace de nexo de unión entre ellos. Los cambios de información pueden ser consultados por los mismos directamente o suscribiéndose y recibiendo una notificación cuando suceda.

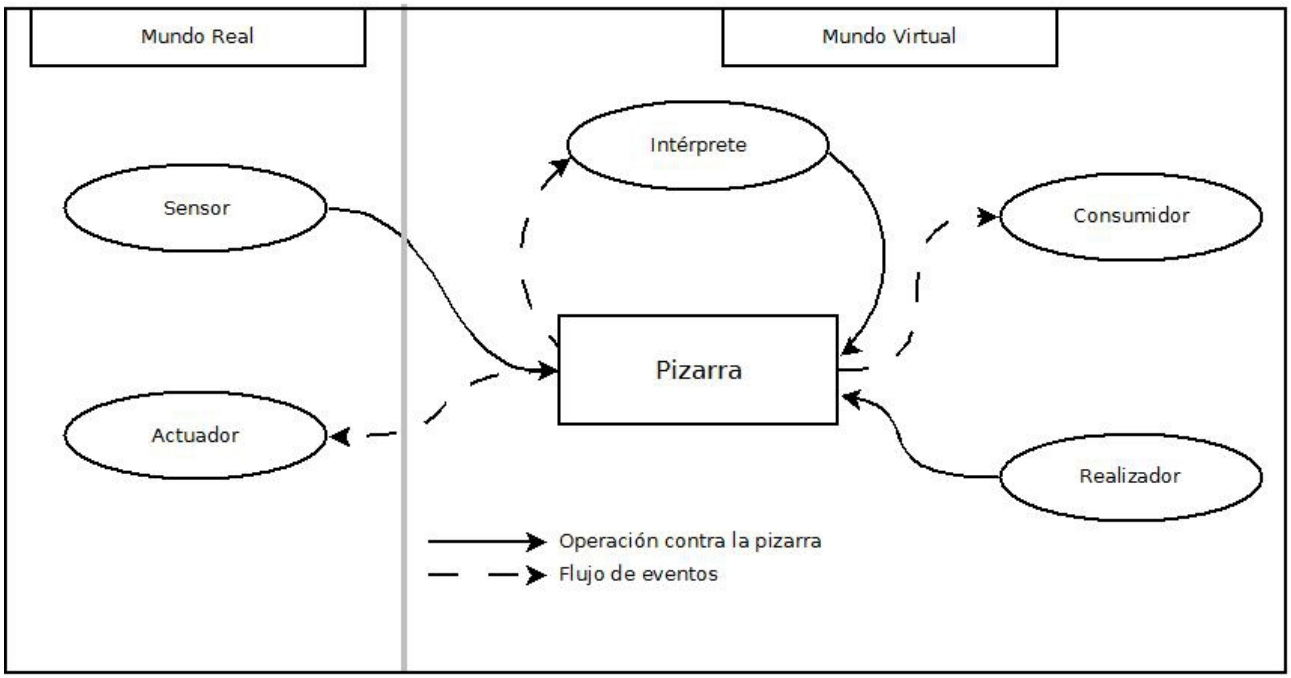


Ilustración 3.4*: Diagrama de comunicaciones usado en la pizarra.*

En nuestro caso, nuestra aplicación será un realizador. Básicamente, se comunicará a la pizarra qué actuador queremos que cambie de estado una vez hayamos tratado el gesto y decidido a qué corresponde.

Realmente se ha elegido éste sistema ya que está montado y comprobado su correcto funcionamiento y no se ha necesitado ningún cambio especial en el mismo, ha sido una reutilización de recursos.

3.3 Catálogo de requisitos del software

A continuación, se presentan los requisitos del software a tener en cuenta durante el análisis, el diseño y el desarrollo. Dentro de los requisitos funcionales, se dividirán en dos secciones:

- **Interfaz.** Se redactarán los requisitos a tener en cuenta en cuanto al diseño y desarrollo de la interfaz con el usuario (tanto visual como de comunicación con la máquina).
- **Librería.** Se indicarán los requerimientos en cuanto al desarrollo de la lógica del proyecto (qué debe hacer, cómo ha de hacerlo, cómo debería estructurarse, cómo de orientada al desarrollador futuro ha de estar, etc.). Además, se comentará cómo ha de ser el movimiento de datos y la estructura de los mismos.

Tras ello, se describirán los requisitos no funcionales los cuales se dividirán en las siguientes subsecciones:

- **Usabilidad.** A qué tipo de usuarios está orientado y la impresión que ha de dar al presentarse al usuario final.
- **Documentación.** Idioma, cantidad, calidad y tipos de documentos necesarios.
- **Mantenibilidad y portabilidad.** Facilidad para, en futuros desarrollos, modificar o portar el código a otros sistemas.
- **Rendimiento.** Aproximación de las necesidades finales del proyecto. Se redactarán los requisitos en cuando a tiempo de respuesta, cantidad de errores por cada acierto, etc.

3.3.1 Funcionales

3.3.1.1 Interfaz: ayuda de calibración

REQ (1) Debe dar la opción al usuario de calibrar la aplicación.

Mostrará una opción al arrancar la aplicación de calibrar el programa. Se implementará por un sistema de botones en el que se decida si calibrar o seguir adelante. En caso de no querer calibrar, el sistema usará la calibración por defecto.

REQ (2) La calibración será guiada por texto.

Antes de proceder a la calibración, el sistema mostrará los pasos a seguir durante la misma. Irá enseñándolo con texto el cual irá apareciendo poco a poco para no saturar al usuario.

REQ (3) La ayuda para la calibración acompañará el texto con imágenes multimedia.

Siempre que se muestre un texto, se acompañará con una imagen o símbolo que ayude a la comprensión. Además, estos símbolos deberán ser parecidos o iguales a los que se usen en la calibración real.

REQ (4) La ayuda a la calibración será fiel a la calibración real.

En caso de que se indique un número, un color o una forma concreta se deberá implementar igual. Solo podrá diferir con la calibración en caso de que no se especifique (v. gr. se indica un cuadrado, pero no tamaño, color o forma; el cuadrado podrá ser más grande en la calibración real).

REQ (5) La aparición del texto tiene que ser fluida para cualquier usuario.

Deberá estar disponible lo suficiente rápido para que un lector habitual no tenga que esperar entre palabras, pero quedar finalmente un tiempo en espera para personas que lean más despacio.

REQ (6) La ayuda finalizará con una situación de éxito.

Se indicará al usuario cómo se finaliza y se le mostrará con algún tipo de datos multimedia (vídeos, imágenes, …) en el que se muestre un ejemplo real de calibración.

REQ (7) La ayuda podrá volver a repetirse las veces que desee el usuario.

Se redirigirá a la misma pantalla de antes de la ayuda o se le preguntará si desea repetirla antes de continuar con la calibración real.

3.3.1.2 Interfaz: calibración

REQ (8) La calibración se repetirá hasta que el usuario acierte el número dado de gestos.

Antes de la calibración se indicará qué número de gestos ha de reconocer acertadamente para que la calibración sea considerada buena. Tendrá que reproducirlos todos antes de poder continuar con la aplicación real.

REQ (9) Habrá, en la calibración, un indicador de si el usuario es reconocido.

Si el usuario es reconocido, se activará el indicador. Mientras tanto, se reservará el espacio o se mostrará el indicador desactivado (en gris, por ejemplo, suele ser una indicación de no activo).

REQ (10) Un indicador mostrará cuándo se ha de comenzar el gesto.

Habrá una imagen parpadeante o un sonido que indique que se ha de llevar a cabo el gesto para comprobar si lo reconoce correctamente.

REQ (11) Si el sistema reconoce un gesto, se le indicará visualmente al usuario.

Si acierta un gesto, al usuario se le mostrará con una imagen, un número o algún tipo de indicación visual que sea reconocible a distancia.

REQ (12) Todas las imágenes y textos de la calibración han de ser reconocibles a distancia.

El usuario no calibrará pegado al monitor. Con lo cual, la interfaz ha de ser lo suficiente grande y las imágenes tener la suficiente resolución como para poder verse a, al menos, un metro o metro y medio de distancia al monitor.

REQ (13) La calibración ha de ser rápida.

No debería llegar a aburrir al usuario. Se ha de conseguir una calibración que, midiendo y calibrando bien, sea lo más rápida posible.

REQ (14) Una vez calibrado, el sistema mostrará el resultado obtenido.

En un futuro, estaría bien que se pudiera leer la calibración de cada usuario de algún archivo de configuración. Para ello, por si acaso, se debería indicar el resultado para poder crear dichos ficheros de configuración.

REQ (15) Una vez calibrado, se dará la opción de ir adelante.

Se mostrará un botón o un enlace a la siguiente parte de la aplicación. Si el botón está visible durante todo el proceso, se desactivará y se activará una vez finalizada.

3.3.1.3 Interfaz: aplicación final

REQ (16) La aplicación reconocerá los gestos del usuario.

Habrá un banco de gestos disponibles y reconocerá todos y cada uno de ellos.

REQ (17) Para poder activar o desactivar un elemento habrá que hacer dos gestos.

El primer gesto indicará a qué elemento del entorno nos referimos. El segundo gesto indicará si hay que apagar o encender, bajar el tono o subirlo, abrir o cerrar... Estos segundos gestos siempre serán iguales independientemente del elemento seleccionado.

REQ (18) Cada elemento tendrá asociado un gesto.

Se ha de mostrar un gesto junto con el nombre o el indicador del elemento.

REQ (19) Los gestos principales aparecerán juntos.

Los gestos se organizaran en la interfaz como en dos secciones dando a entender que no pertenecen a lo mismo (ya que, si haces los secundarios sin principal, no debería suceder nada).

REQ (20) Cuando un elemento esté seleccionado, se indicará al usuario.

Se recuadrará el gesto, iluminará o hará cualquier cosa que se considere necesaria para que sea notable que el elemento se ha escogido. Se deberá diferenciar de los demás sin escoger y mantener activo mientras no se seleccione otro elemento.

REQ (21) Aunque se haya hecho el gesto secundario, se mantendrá el elemento activo.

Se podrá volver a hacer otro gesto secundario en el gesto aunque ya se haya hecho uno primero (v. gr. cuando se sube la luz en una luz graduable se requiere que sea continuo, no sería eficiente tener que hacer ocho gestos, cuatro primarios y cuatro secundarios, cuando puedes realizarlo con cinco, uno primario y cuatro secundarios).

REQ (22) La aplicación solo podrá ser usada por una persona simultáneamente.

Al menos de momento, se restringirá a una persona para evitar problemas con los gestos. Como comentaremos más adelante, se tendrá en cuenta para un trabajo futuro.

3.3.1.4 Librería: conexión de la Kinect

REQ (23) Abstraerá al usuario de la conexión de la Kinect.

No se quiere que el desarrollador que use la librería tenga que debatirse con el API de la Kinect. Se tratará de crear una librería que gestione todas las funcionalidades necesarias de la misma.

REQ (24) Se ha de generar un código inteligible y altamente comentado.

Se requiere que el código sea inteligible y comentado para abrir la puerta a futuros desarrollos o, incluso, que varios desarrolladores puedan cambiar el código sin la necesidad de un aprendizaje previo.

REQ (25) Tendrá dos métodos principales: de inicialización y parada.

Para abstraer al futuro desarrollador, se crearán dos métodos (al menos) con los que se inicie y pare la Kinect. El de inicio comenzará todos los servicios necesarios para la librería y el de parada matará los procesos y liberará la Kinect para otros usos.

REQ (26) Guardará la función que recoja y evalúe los puntos.

Como se desea que la librería se pueda usar para una gran variedad de proyectos, se necesita que la función que evalúe los puntos se pueda asignar desde fuera de la misma. Por ello, la librería la guardará y asignará automáticamente.

3.3.1.5 Librería: los gestos

REQ (27) Los gestos que reconozca se dividirán en movimientos.

Se tratará de hacer altamente personalizables los gestos para que cada desarrollador pueda usar los que necesite. Para ello, se dividirán los gestos en movimientos, los cuales serán un vector que indique hacia donde se ha de mover la mano.

REQ (28) Ambos gestos y movimientos se definirán en un archivo XML.

Cada gesto o movimiento tendrá un XML asociado. Su facilidad de lectura y de modificación es la razón de esta decisión.

REQ (29) Los movimientos tendrán definido un rango de error.

Cómo todo hardware, la Kinect no es perfecta. Se ha de plantear un sistema de reconocimiento del movimiento que reconozca el movimiento dentro del rango de error que se defina. Aún así, el sistema no se planteará para que haya más de un movimiento en un ángulo determinado. Será competencia del desarrollador definir unos movimientos únicos para cada ángulo o, en caso de que se desee, no definir ninguno.

REQ (30) Los gestos sabrán los movimientos que necesita a través de un identificador.

Será competencia del futuro desarrollador seleccionar un ID para cada movimiento que luego se pueda relacionar con un gesto. El ID tendrá que ser único y en caso de no serlo tendrá un funcionamiento no definido.

REQ (31) Cada gesto también tendrá asociado un identificador.

Este ha de ser único (pero no con respecto a los movimientos, son identificadores diferentes). Este ID será elegido por el desarrollador ya que, en caso de encontrar el gesto, será el identificador que use la librería. Será competencia del futuro desarrollador que el ID no se repita. No se especifica resultado en caso de repetición.

REQ (32) Cada gesto tendrá la posibilidad de definir unas propiedades.

No se especifican las propiedades de momento, pero se deja abierto para un trabajo futuro. V. gr. que los movimientos del gesto sean todos iguales (en tamaño).

REQ (33) Se creará un fichero de configuración de la Kinect y los gestos.

V. gr. los gestos han de tener un número definido de movimientos. Este dato y los que se necesiten se definirán en ese fichero de configuración.

REQ (34) Siempre existirá el «gesto vacío».

Se reservará el ID 0 al mismo. No habrá que generar un XML para el mismo.

REQ (35) Los movimientos se definirán con un vector en dos dimensiones.

Como la Kinect devuelve tres dimensiones, se creará un sistema para abstraer estas tres dimensiones en dos. Se considerará que el plano donde tomar las dos dimensiones será el plano que genere el vector con el suelo (el vector del plano que forme menor ángulo con el vector del movimiento).

3.3.1.6 Librería: rastreador

REQ (36) El rastreador se encargará de la gestión de los gestos.

Se abstraerá del uso de la Kinect para recibir los puntos de manera que, en un futuro, se pueda seleccionar el hardware que se prefiera para poder hacer uso del rastreador.

REQ (37) Al inicio, el sistema guardará y configurará todos los gestos y movimientos.

Se definirá una carpeta con los gestos y una con los movimientos. Al inicio se comprobarán las mismas y el sistema almacenará todos los que se hayan definido correctamente.

REQ (38) Todo gesto, movimiento o propiedad generará un objeto.

Para una mejor gestión, se crearán unas clases donde almacenar los datos mientras la Kinect esté en funcionamiento. Cada gesto, movimiento o propiedad leída será asignada a su clase correspondiente al cargarse por lo que solo se leerán una vez. Además, como no se sobrescribirán los XML desde la librería, no se necesita su lectura posterior.

REQ (39) La gestión de los puntos se hará a través de una sola función.

Esta devolverá un código de error en caso de fallo, otro en caso de que no sea el momento de rastrear el punto, cero si no ha reconocido gesto y el identificador del gesto en caso afirmativo.

REQ (40) De todos los puntos se seleccionarán los necesarios según un rango de tiempo.

El tiempo se definirá según llegue de la calibración o, en caso de que no llegue, se introducirá uno por defecto. Los puntos intermedios desde que se recibe uno hasta el siguiente no serán tratados.

REQ (41) Para reconocer un gesto se han de reconocer todos los movimientos intermedios.

Se comprobarán los movimientos en el orden de llegada y solo devolverá un gesto en caso de que todos los movimientos de ese gesto se hayan encontrado y en el orden indicado. Si no, devolverá el gesto vacío.

3.3.2 No funcionales

3.3.2.1 Usabilidad

REQ (42) Un usuario medio debe entender el funcionamiento sin problemas.

Será imprescindible que la interfaz sea agradable y fácil de usar, que no incluya ningún elemento contradictorio y no necesite demasiadas explicaciones.

REQ (43) Los gestos han de ser similares y simples.

Es una aplicación para un uso general. No se quiere que los usuarios necesiten aprender una gran cantidad de gestos o mayor explicación que una imagen.

3.3.2.2 Documentación

REQ (44) Todo el código estará comentado.

Cada función, clase, variable, etc., llevará una cabecera con los datos que luego autogenerará la documentación.

REQ (45) Se llevará a cabo una documentación escrita sobre el funcionamiento.

Habrá que escribirla en español e inglés. Incluirá funciones básicas, propiedades, ejemplos...

3.3.2.3 Mantenibilidad y portabilidad

REQ (46) Solo se desarrollará para sistemas Windows.

Como hemos comentado anteriormente, la API oficial solo se encuentra desarrollada en Windows. En caso de que Windows sacara una versión para Linux, se planteará la portabilidad.

REQ (47) Se plantea un mantenimiento de al menos un año.

Tras ese periodo se tratará de actualizar el sistema y hacerlo compatible con la futura Kinect. Se dispondrá un 25% del tiempo dedicado al mantenimiento en mantenimiento correctivo, un 60% en mantenimiento perfectivo, un 15% en mantenimiento adaptativo y un 5% en preventivo.

3.3.2.4 Rendimiento

REQ (48) La aplicación reconocerá al menos a un usuario.

De momento se deja como trabajo futuro el reconocimiento de gestos con más de un usuario en el rango de visión.

REQ (49) El tiempo de respuesta debe de ser menor o igual a 2 segundos.

Desde que el usuario comienza el gesto hasta que el sistema lo reconoce, lo transmite a la pizarra y esta al actuador.

REQ (50) El sistema no sobrepasará los 2 GBs de RAM.

Uno de los mismos será el usado por el sistema operativo y se reservará el otro para nuestra aplicación.

3.4 Diseño de la arquitectura del sistema

Durante el desarrollo, como se expondrá más adelante, se tuvo que cambiar la visión del proyecto. Hubo que reestructurarlo para poder combatir ciertos problemas que comenzaron a surgir tales como la depuración de la aplicación. Con lo cual, se cambió gran parte del diseño de la misma, y todos los componentes de su alrededor.

En los próximos apartados se detallá el diseño del sistema, pero tras ese cambio. Se han omitido los cambios que supuso ya que en algunos casos eran nimios y en otros realmente hubo una reestructuración completa. En conclusión, no aportará al proyecto más información el redactarlo.

3.4.1 Diagrama de componentes y sus interfaces

Se ha decidido reducir el sistema completo a cuatro componentes generales, algunos de los cuales se han subdividido acorde a las necesidades. A continuación, se muestra un diagrama de componentes UML y sus interfaces (**Ilustración 3.5**) que, aunque bien se puede usar para describir componentes físicos, se ha visto aclarador el uso de los mismos para detallar el funcionamiento. En estos diagramas cada caja corresponde a un componente, cada cuadrado en su borde a un puerto (de entrada o salida), cada recta finalizada con una circunferencia a una interfaz de salida y las acabadas en una semicircunferencia a una interfaz de entrada.

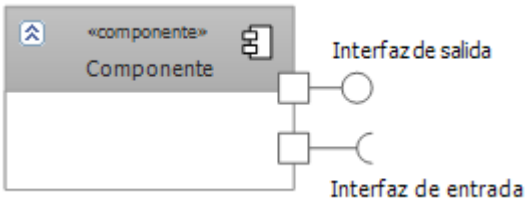


Ilustración 3.5*: Elementos de un diagrama de componentes UML.*

El sistema tendrá todo su trabajo de computación real en lo que se ha decidido llamar **librería**. Esta será la encargada de poner en funcionamiento a nuestro dispositivo, la Kinect, además de hacer el cálculo de los gestos. Se comunicará directamente con la **interfaz**, que necesitará del paso de parámetros y llamadas de la misma a la librería y esta última le devolverá toda la información requerida (puntos, gestos y diferentes tipos de datos extraídos del Kinect).

La **interfaz** será el componente encargado de gestionar la GUI (*Graphical User Interface*). Mostrará al usuario por pantalla los datos e indicaciones que se necesiten para la ejecución de la aplicación. Además, se comunicará con la **librería**, de la que sacará la información que vaya aportando el usuario con la Kinect. Comprobará esta información y, en caso de que se encuentre que el usuario ha ejecutado un gesto de los dispuestos en la GUI, enviará la misma a la **demo**.

El tercer componente de nuestro sistema es la **demo**. Es el menor de todos y el que menos implicación tendrá en el desarrollo, pero es totalmente necesario. Una vez le lleguen los datos de la **interfaz**, se encargará de comprobar ese dato. Se creará una interfaz entre demo y pizarra la cual se tendrá que llevar a cabo para tener una comunicación con el entorno sin tener que detenerse a desarrollar nuevas librerías en C# (se encuentra actualmente en Java). Asociará cada valor recibido con una acción dentro de la **pizarra** (v. gr. si le llega un 1 indicará que se apague una luz). Tras ello, evaluará y mandará la acción a la pizarra ya tratada. Este componente sería el que, como indicamos en el **capítulo 3.2.4**, sirva de realizador.

El cuarto y último componente será la **pizarra**. Este sistema, como hemos descrito, ya ha sido desarrollado dentro del entorno de trabajo, el AMILAB, por ello no nos detendremos en exceso a la descripción de la misma. Simplemente comunicará la **demo** (el realizador) con el actuador final que se haya indicado. Se han decidido de momento actuadores simples tales como luces y puertas, con valores binarios, y uno un poco más complejo como una lámpara regulable.

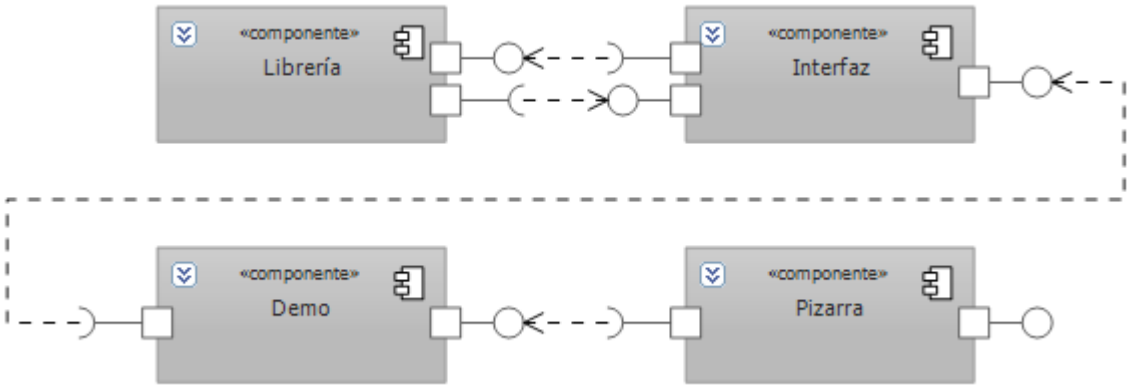


Ilustración 3.6*: Diagrama UML de componentes general.*

Entrando más en detalle, los componentes más importantes del proyecto serán la **librería** y la **interfaz**. Cada una se dividirá en componentes más pequeños para un tratamiento más eficiente de la información, se fragmentará de tal manera que sea más fácil el uso y modificación de las mismas además de permitir implementar nuevas funcionalidades que desee el futuro desarrollador.

La **librería** tendrá dos componentes internos (**Ilustración 3.7**):

- El **Runner** (lanzador) se encargará de correr la Kinect. Configurará los parámetros necesarios, encenderá los sensores que se utilicen y la situará en espera. Todas las funciones necesarias en el desarrollo que impliquen el uso de la Kinect para recoger datos pero que no necesiten de los puntos serán implementadas en este componente. Dispone una interfaz de entrada, donde recibirá la información y peticiones, y otra de salida.
- El **Tracker** (rastreador) tendrá la función de comprobar los puntos. Organizará los mismos, eliminará los no deseados y comprobará los gestos. Será el punto fundamental de la librería. Se requiere la división con el Runner ya que, como se ha detallado anteriormente, se ha de implementar abstrayéndose del dispositivo que usemos de entrada. Tendrá una interfaz de entrada y una de salida. Llegarán puntos y devolverá el resultado.

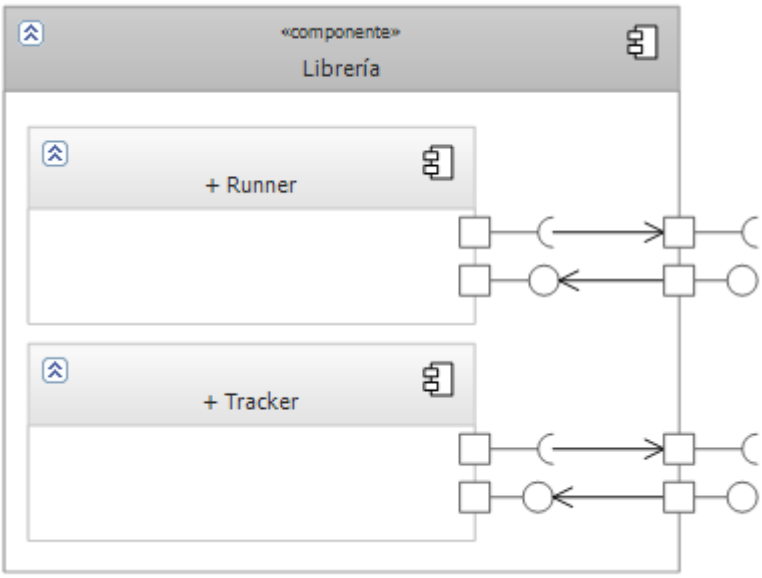


Ilustración 3.7*: Diagrama UML de componentes de la librería.*

La **interfaz** también se dividirá en dos componentes (**Ilustración 3.8**), aunque esta vez no son tan independientes como la anterior:

- El **Calibrador** será lo primero que ejecute el usuario (si así lo desea). Se iniciará con un gesto predefinido, y tratará de ajustar el tiempo de cada movimiento del usuario. Se comunicará con la librería para recibir los puntos y comprobar si los mismos llevan a cabo un gesto. Tendrá una interfaz de entrada y una de salida para este propósito. Además, dispondrá de otra interfaz de salida para que, en caso de ejecutarse y finalizar la calibración, se le indique al general el tiempo de movimiento.
- El **General** tendrá el propósito ya de la aplicación en sí: recibir los puntos, comprobar los gestos y, en caso de que este se corresponda con alguno de los que se han definido para el uso, enviarlos a la **demo** para su uso. Tendrá, al igual que el Calibrador, una interaz de entrada y otra de salida con la librería. Además, tendrá la interfaz de entrada que se comunicará con el anterior para obtener el tiempo de movimiento. Finalmente, necesitará de otra interfaz de salida por donde enviará los datos a la demo.

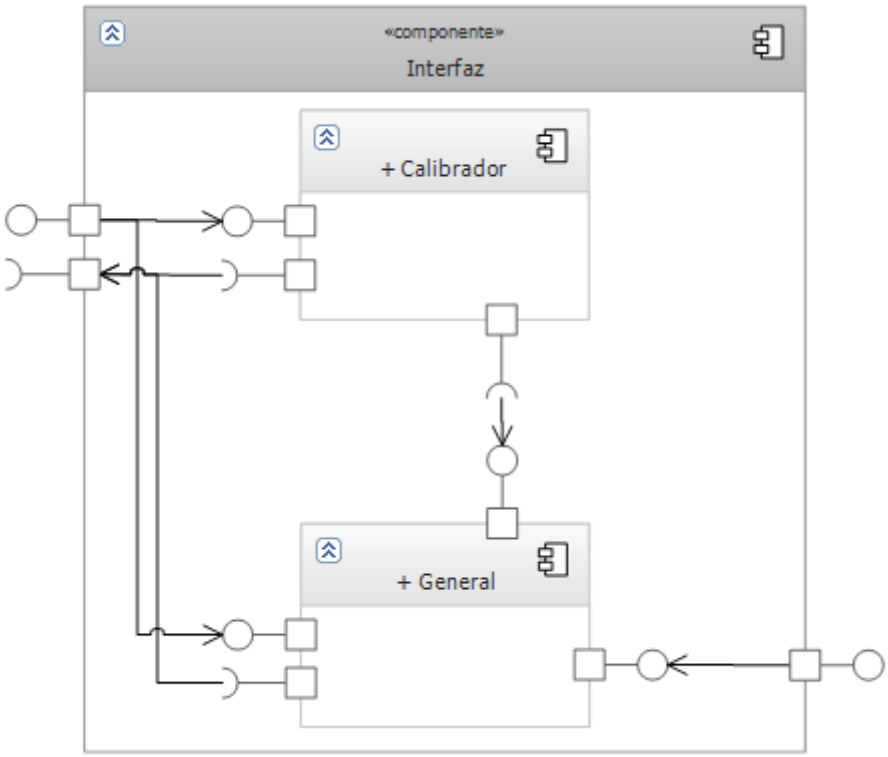


Ilustración 3.8: *Diagrama UML de componentes de la interfaz.*

En la **interfaz**, como veremos en el **capítulo 3.5**, no se dará demasiado peso a un desarrollo logrado. Se requiere que sea útil y fácil, pero sale fuera de los términos del proyecto el desarrollar una interfaz con las últimas tecnologías que, además, sea vistosa.

3.4.2 Interacción entre los componentes

En esta sección se va a tratar de definir el comportamiento del sistema (visto desde los componentes **librería** e **interfaz**) en tres diferentes escenarios.

Los dos primeros serán la calibración y una ejecución normal. Partimos de la base de que el control de los gestos se realiza correctamente y en caso de la calibración se devuelve un *true* o *false* si se ha reconocido el gesto o no o en caso de la ejecución general, devuelve el gesto reconocido. Como se comentó en el **capítulo 3.3.1.5**, se asumirá que el caso de gesto vacío se controla como un gesto de ID igual a cero.

También se definirá el proceso de comprobación de un gesto, el cual es común para los anteriores escenarios. Para ello, se tomará un ejemplo con dos gestos y dos movimientos por gesto.

3.4.2.1 Escenario 1: calibración

A lo largo de la calibración, se requerirán de los dos componentes principales del sistema: la **librería** y la **interfaz**. Se partirá de la situación en que el usuario ha iniciado la aplicación y se dispone a calibrar (a presionado sobre el botón o enlace de calibrar). A partir de esa situación, los pasos aproximados que toma nuestro escenario serían (**Ilustración 3.9**):

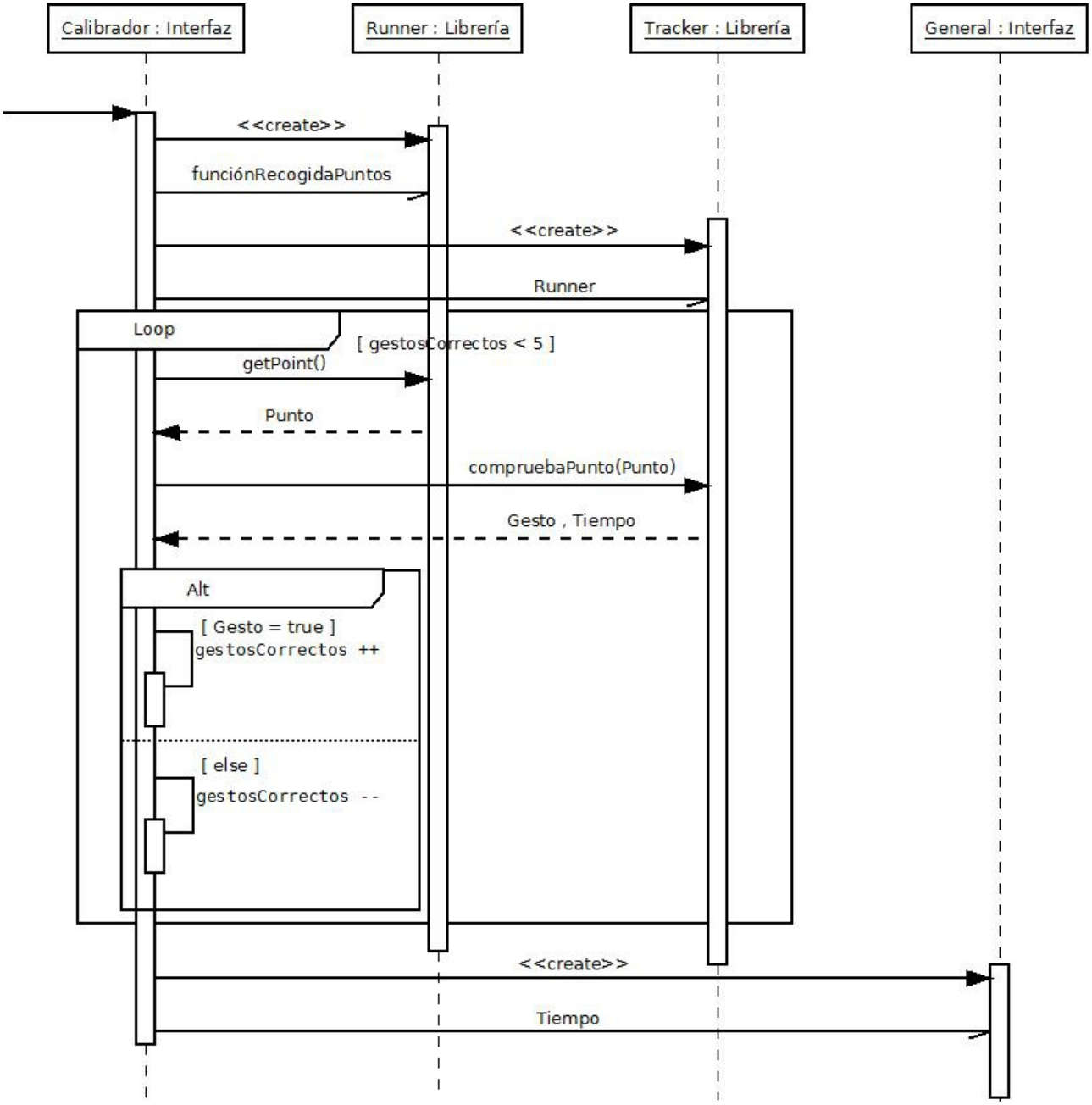


Ilustración 3.9*: Diagrama de secuencia de calibración.*

- Creación y configuración del **Runner** y el **Tracker**. Además, al Runner se le asignará la función que se desee para el tratamiento de los puntos (la cual estará implementada en la **interfaz**) y al Tracker se le asignará el Runner para ciertos parámetros de configuración.
- La función que se ha asignado al Runner será llamada cada vez que haya un nuevo conjunto de puntos disponible. De este conjunto, se usará solo el punto seleccionado (de momento será solo la mano derecha) y se le mandará al Tracker para su comprobación.
- El Tracker comprobará si ese punto es un punto válido (el Runner devuelve más puntos de los que se requieren, por lo tanto se eliminarán los intermedios) y, en caso afirmativo, lo incluirá en los puntos comprobados. De estos puntos, el Tracker comprobará si en ese momento se genera un gesto o no. Si lo hace, se devolverá el gesto reconocido, si no, el gesto vacío.
- Una vez se reconozcan cinco gestos aproximadamente seguidos, se eliminarán el Runner y el Tracker y se llamará al componente general con el tiempo resultante.
- Durante todo el proceso se irá incrementando o decrementando el tiempo de reconocimiento de los movimientos. Cuando devuelva el quinto gesto, el tiempo asociado será el resultado.

3.4.2.2 Escenario 2: ejecución normal

Una vez calibrado (si así lo ha deseado el usuario) y recogido el tiempo resultante, se volverá iniciar el proceso, ahora orientado a una **ejecución normal** (**Ilustración 3.10**). Partimos de la base de que tenemos ese tiempo y que la aplicación se ha iniciado. Se asumirá que el usuario conoce los gestos (la GUI se encargará de mostrárselos) y que el tiempo de movimiento es el perfecto para él. También se encontrará la **demo** y la **pizarra** corriendo y en perfecto funcionamiento para poder llevar a cabo la petición.

En este caso, los pasos aproximados de ejecución serían:

- Al igual que en la calibración, se han de crear Runner y Tracker. Esta vez, la única diferencia es que al Tracker se le asignará un tiempo antes de comenzar para calcular los movimientos.

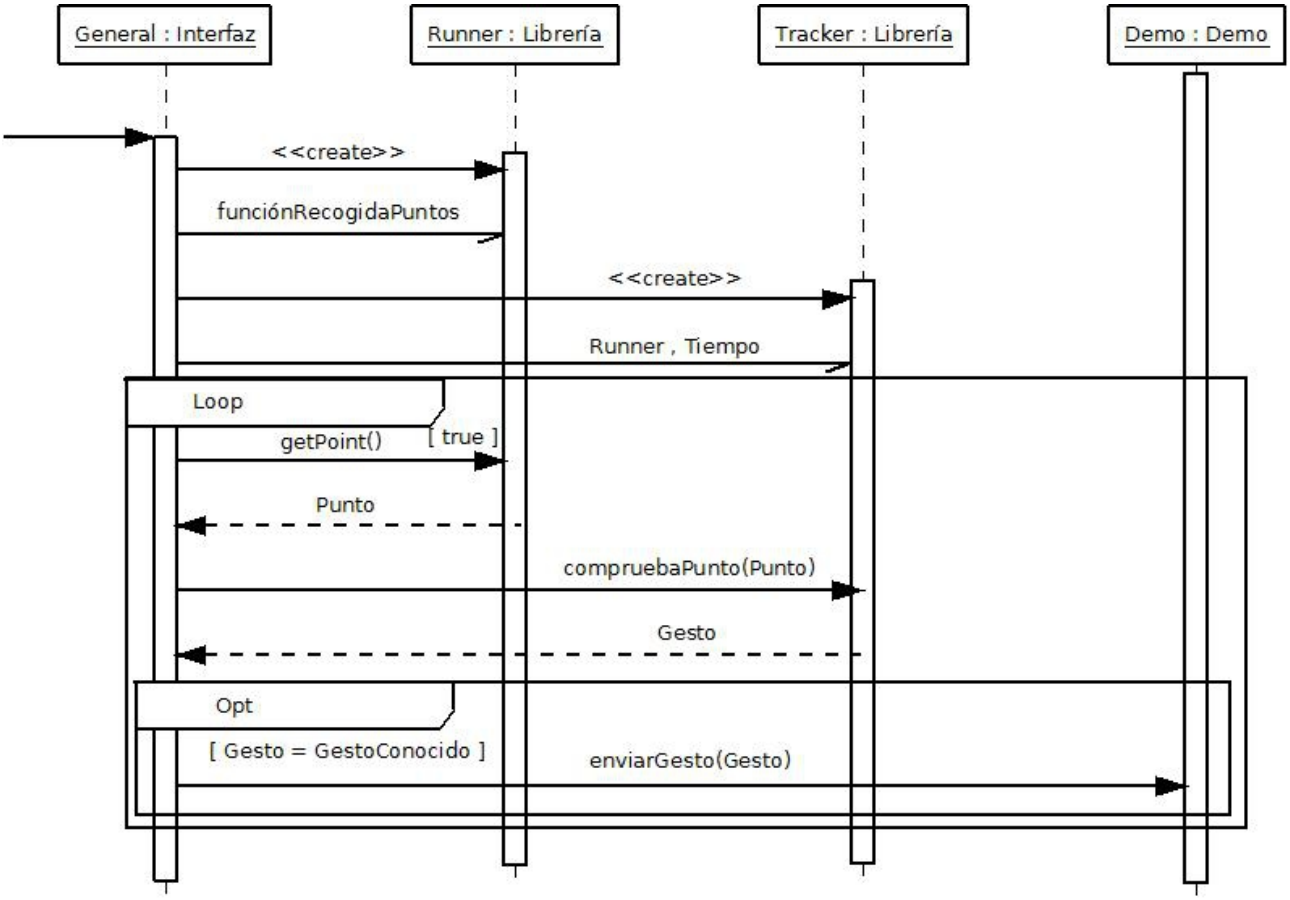


Ilustración 3.10*: Diagrama de secuencia de ejecución normal.*

- Parecido al anterior, la función que hayamos entregado al Runner obtendrá los puntos y se los enviará al Tracker para su comprobación. Este también devolverá el gesto obtenido en caso de acierto.
- Finalmente cambia la ejecución. En este caso, si hemos encontrado un gesto conocido, que tiene una función dentro de la aplicación, será enviado a la **demo** que, como se comentó anteriormente, se encargará de transformarlo y enviarlo a la **pizarra**.

El proceso, como se puede comprobar, será más simple en el caso de la ejecución. No necesitaremos llevar la cuenta de gestos reconocidos ni guardar el tiempo ya que solo se usa inicialmente. La gran diferencia será el número de gestos a tratar y cómo asignarlos. En la calibración se asume un solo gesto, aunque podría hacerse con otros, mientras que en esta última se deberán reconocer todos los gestos que haya definido el desarrollador.

3.4.2.3 Escenario 3: comprobación de un gesto

Para el siguiente escenario se ha tenido en cuenta una situación en la que se hayan definido dos movimientos y dos gestos (estos como máximo se pueden componer de dos movimientos en esta situación). Además, se ha supuesto que el usuario ha iniciado la aplicación y se han cargado todos los datos. Por último, se ha asumido que siempre reconoce un gesto. Se puede dar el caso contrario. En ese caso, el programa no seguirá comprobando el gesto y devolverá el gesto vacío.

A continuación, se presenta un diagrama de secuencia del mismo (**Ilustraciones 3.11 y 3.12**) acompañado de una explicación de cada uno de los procesos más esenciales.

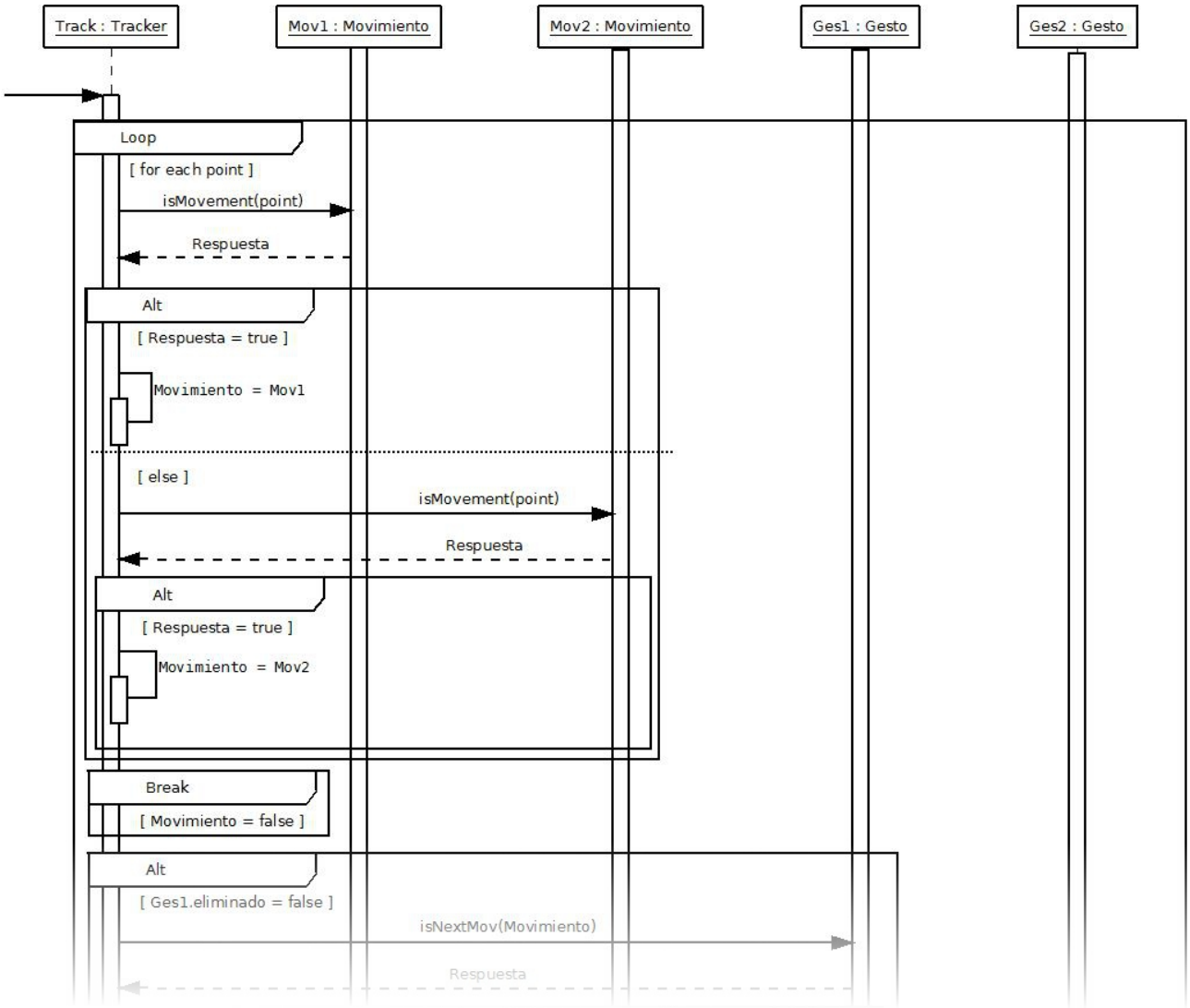


Ilustración 3.11: *Diagrama de secuencia de comprobación de un gesto (1).*

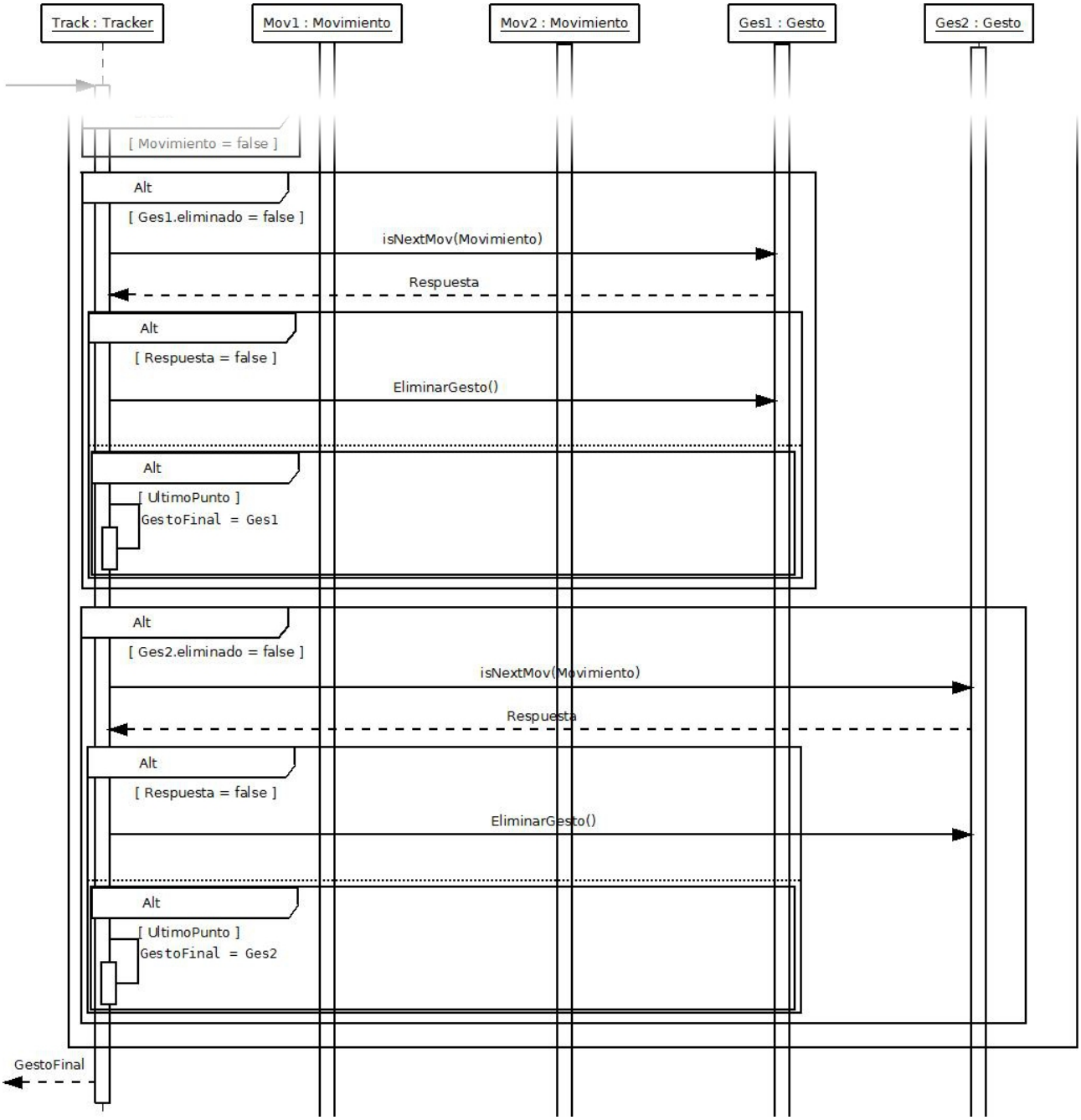


Ilustración 3.12*: Diagrama de secuencia de comprobación de un gesto (2).*

El proceso se dividirá, básicamente, en dos fases. En la primera se recorrerán los movimientos en busca de uno que encaje con el vector entregado. De segundas, una vez recuperado en caso de existir, se comprobarán los gestos disponibles. Si el movimiento no es el siguiente del gesto, este se eliminará junto a sus comprobaciones. Una vez recorridos los gestos, el sistema solo debería devolver uno (el cual tenga todos los movimientos comprobados y en el orden entregado).

3.5 Diseño visual de la interacción con el sistema

En el apartado en el que nos encontramos vamos a comenzar a describir la interfaz de usuario de la futura aplicación. Aunque la librería se vaya a poder usar en cualquier desarrollo, como se comentó anteriormente se va a crear una pequeña demo para la comunicación con el entorno la cual contiene la siguiente GUI.

El apartado se dividirá en dos subapartados: uno dedicado a la calibración y otro a la aplicación general. Se mostrarán dos maquetas, una por subapartado de las cuales se describirá el flujo (se señalará con números en rojo), los elementos que tiene (se indicarán con números en verde), y la interacción del usuario con ella.

3.5.1 Interfaz de usuario de la calibración

Empezando por la calibración, la interfaz propuesta se ve reflejada en la **ilustración 3-13**. Se dividirá en siete pantallas con sus respectivas transiciones en las mismas. La mayoría de ellas será activada sin acción alguna del usuario. Además, cuenta con seis elementos diferentes dentro de la misma.

El flujo de la interacción de usuario, como se explicó con anterioridad, viene señalado con números en rojo y sería el siguiente:

0. Una vez iniciada la aplicación, el usuario se encontrará con dos botones. Uno de ellos le llevará a la calibración[1] y el otro será el encargado de redirigir a la aplicación real[2]. Para seguir con la calibración, presionaremos el primer botón.

1. Tras presionarlo, comenzará un tutorial que irá avanzando automáticamente. En esta segunda pantalla nos encontraremos nada más que un cuadro de texto[3]. Este incluirá una presentación a la ayuda y una descripción de la misma. En cuanto aparezca todo el texto y pase un pequeño intervalo, se redirigirá al usuario a la siguiente pantalla.

2. En la pantalla actual, la tercera, nos encontramos con tres elementos diferentes. El primero es una imagen que indica si el usuario se encuentra frente a la Kinect[4] y esta le ha reconocido. Si se cumple, la imagen será verde. Si no, gris. El segundo será una imagen intermitente que indicará al usuario cuando ha de hacer el gesto[5]. Finalmente, encontraremos un nuevo cuadro de texto que explicará los elementos superiores.

3. Una vez se haya explicado todo, volverá a avanzar la película. Esta vez habrá un nuevo elemento y el cuadro de texto se habrá vuelto a reducir. Este nuevo elemento es una tira de imágenes que indicarán al usuario el número de elementos que ha reconocido de una manera visual[6]. El texto describirá ese nuevo elemento. Una vez descrito, al igual que anteriormente, volverá a avanzar el tutorial.

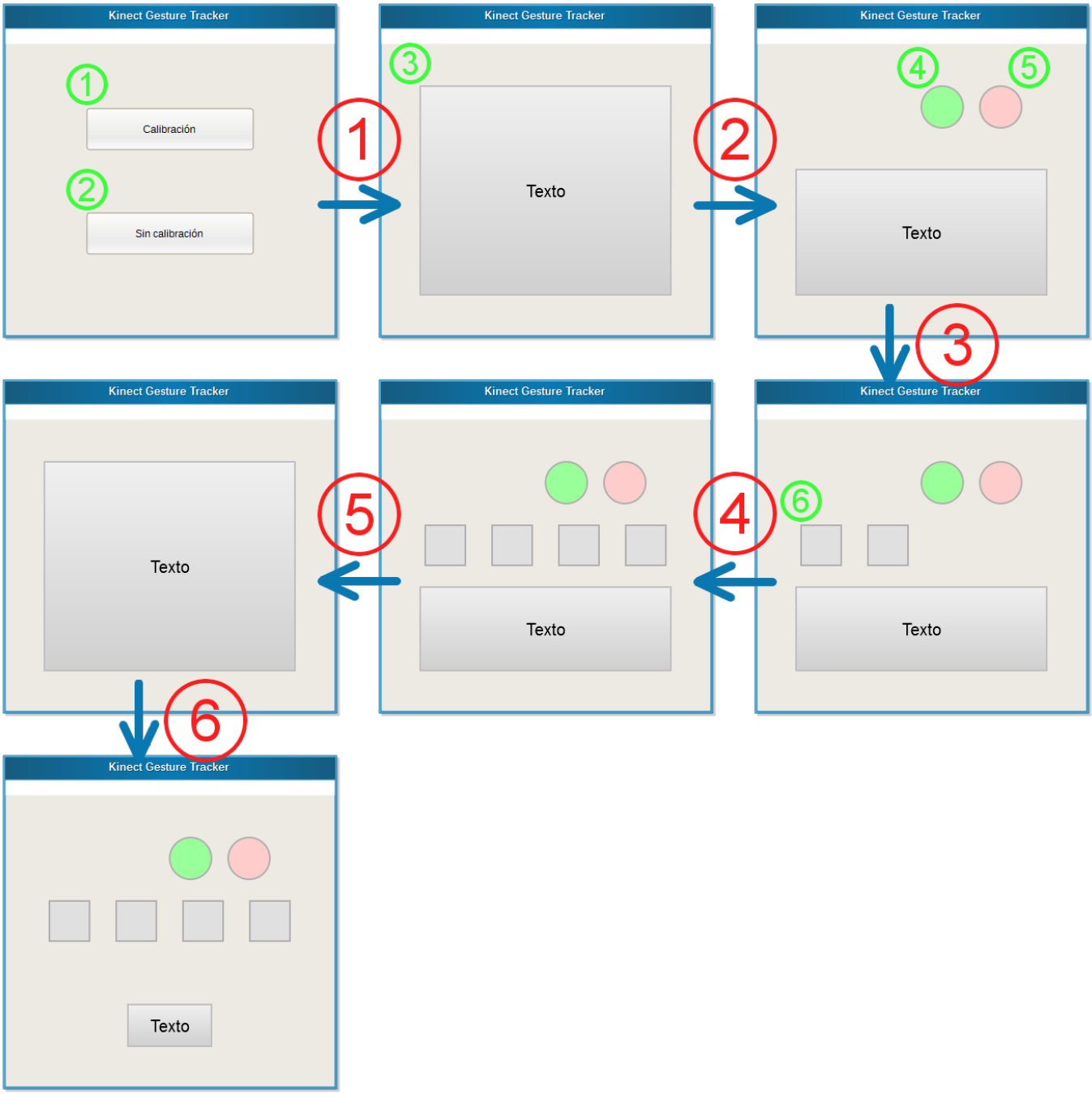


Ilustración 3.13: *Diseño visual de la calibración.*

4. En esta nueva pantalla solo se mostrará el resultado final de la calibración, cómo debería percibirla el usuario para llegar a la conclusión de que ha finalizado. Al igual que en todas las pantallas anteriores, se mostrará de forma gráfica y usará un cuadro de texto para comentar los detalles necesarios.

5. Se mostrará únicamente, igual que en la segunda pantalla, un cuadro de texto. Despedirá al usuario y le indicará los últimos detalles sobre la calibración.

6. Finalmente, tras terminar de mostrar el texto, el sistema redirigirá al usuario a la calibración real. Aquí el cuadro de texto será algo simple, que no distraiga al usuario. La información que mostrará será el tiempo por movimiento actual. Se irá actualizando por cada nuevo gesto pedido al usuario y, una vez el usuario cumpla con el reto, se mantendrá fija con el resultado final (el cual, en un futuro, se podrá usar para crear configuraciones personalizadas).

	Elemento	Descripción
1	Botón	Redirigirá a la calibración al pulsar.
2	Botón	Redirigirá a la aplicación real (sin calibrar) al pulsar.
3	Cuadro de texto	Mostrará el texto de la pantalla. Aparecerá poco a poco, haciendo algún tipo de efecto gráfico. Todos los cuadros con texto actuarán igual.
4	Imagen / Imágenes	Indicará cuando el usuario se encuentra frente a la Kinect. Para ello, se pondrá en verde en ese momento o en gris si no cumple las condiciones.
5	Imagen / Imágenes	Indicará al usuario cuando debe ejecutar un gesto. Estará en gris siempre y parpadeará a rojo cuando sea necesario.
6	Imagen / Imágenes	Tira de imágenes de cuadrados. Cada vez que aparezca un cuadrado significará que el usuario ha ejecutado bien el gesto y la Kinect lo ha reconocido. Si ya hay algún cuadrado y el usuario falla, desaparecerá el último.

Tabla 3-2*: Elementos visuales de la calibración.*

Una vez finalizada la calibración, el sistema redirigirá al usuario a la aplicación real. El flujo de actividad será el mismo que el mostrado en el próximo subapartado, con la diferencia de no tener que presionar el botón inicial para llegar a él y que el tiempo resultante de la calibración será entregado para iniciar el Kinect con la personalización.

3.5.2 Interfaz de usuario de la aplicación real.

La interfaz de usuario de la aplicación real es, realmente, muy pequeña, no tiene muchos datos en la misma. Trata de mostrar al usuario simplemente los gestos acompañados de una pequeña descripción.

El sistema de colores en la **ilustración 3-14** es igual que en el apartado anterior. Una interacción normal con el usuario sería:

0. Al igual que en la anterior, el usuario comenzará iniciando la aplicación. Esta vez en vez de presionar el botón de calibrar[1], tendrá que presionar el botón de «Sin calibración»[2]. Una vez lo presione, la aplicación avanzará a la aplicación real.

1. En esta pantalla, encontraremos divididas dos partes. Arriba se encuentran los gestos necesarios para seleccionar un elemento (con una imagen[3] del mismo y su identificador[4]) y abajo los gestos necesarios para ejecutar una acción sobre ellos (al igual, una imagen[5] y un texto describiendo la acción[6]). El usuario decidirá el elemento sobre el que actuar y ejecutará el gesto para pasar a la siguiente pantalla.

2. El único cambio real que percibirá el usuario será una iluminación detrás del elemento sobre el que quiere actuar[7]. Esto le indicará que se ha reconocido el gesto y ha reconocido el que el usuario deseaba. Por último, el usuario podrá ejecutar una acción de las presentadas en la zona inferior realizando el gesto correspondiente. La interfaz no mostrará el cambio, pero podrás comprobarlo por el cambio en el actuador.

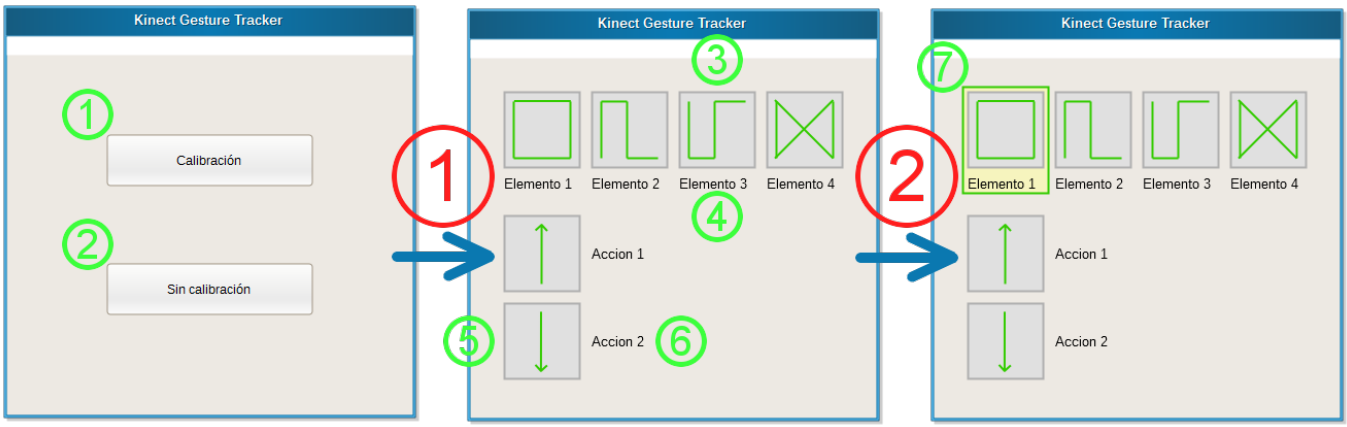


Ilustración 3.14: *Diseño visual de la aplicación general.*

	Elemento	Descripción
1	Botón	Redirigirá a la calibración al pulsar.
2	Botón	Redirigirá a la aplicación real (sin calibrar) al pulsar.
3	Imagen	Mostrará una imagen del gesto asociada al elemento del entorno.
4	Texto	Nombre del elemento del entorno asociado a la imagen superior.
5	Imagen	Mostrará una imagen del gesto asociada a la acción a realizar.
6	Texto	Descripción de la acción asociada a la imagen de la izquierda.
7	Color (v. g. imagen detrás).	Fondo de color que indicará que el elemento detrás del que se encuentra es el que se encuentra activo.

Tabla 3-3*: Elementos visuales de la aplicación real.*

CAPÍTULO 4 IMPLEMENTACIÓN

En este capítulo se procederá a describir el proceso de implementación del sistema, su resultado final y los diferentes problemas que se han ido encontrando por el camino y cómo se ha conseguido atajarlos, si es el caso, o por qué se mantienen.

Se dividirá su contenido en tres apartados principales, las dos implementaciones que se han llevado a cabo y uno pequeño explicando los cambios que se han llevado y la razón de los mismos. En el primero se comentará lo que se desarrollo hasta el momento de decidir cambiar el desarrollo. En el segundo, los cambios y su razón. En el tercero ya se detallará el desarrollo final.

Se dividirá por componentes, como los presentados en el capítulo de análisis y diseño. También se fragmentarán estos componentes y se detallará el contenido de los mismos, su funcionamiento y otra información importante.

4.1 Primera implementación

A grandes rasgos, esta primera implementación constaba de dos de los componentes del sistema: **interfaz** y **librería**. La **demo**, al ser realmente una aplicación intermedia para comunicar la **pizarra** con el resto del sistema, se decidió dejar para el final, no se necesitaba hasta que no se tuviera al sistema reconociendo gestos de una manera más o menos fiable.

La gran diferencia entre esta implementación y la final recae sobre la librería. Hubo que reestructurar el reconocimiento de los gestos. Como detallaremos a continuación, se trató de llevar a cabo un árbol de estados lo cual dificultó mucho al hacer depuración del programa sobre todo.

A grandes rasgos, esta librería recogía todos los puntos y los evaluaba punto a punto. El problema surgió cuando la Kinect, al no ser perfecta, devolvía un punto extraño. Por culpa de ese punto, el sistema te sacaba normalmente del estado en el que te encontrabas. Nunca se llegó a conseguir encontrar un gesto completo, solo encontraba gestos atómicos, movimientos.

4.1.1 Interfaz

El desarrollo de la interfaz es algo que se decidió dejar para una segunda iteración. En la que se encontraba, solo se usaba una pequeña interfaz donde se mostraban los movimientos. Se tomó como referencia el ejemplo *Skeleton Basics* del *Developer Toolkit Browser* [34] que aporta Microsoft. Se disminuyó el tamaño del cuadro donde se muestra el esqueleto y encima del mismo se añadieron unas imágenes que se activaban cuando se ejecutaba un movimiento o un gesto.

No se desarrolló un calibrador. Con la estructura que se comenzó no era necesario.

4.1.1.1 General

A continuación se presenta un ejemplo de la interfaz de pruebas. No sería una interfaz definitiva, el diseño de la interfaz fue el mismo para la primera implementación que para la segunda, pero se demuestra cómo se fueron haciendo las pruebas iniciales.

En la primera imagen (**ilustración 4-1**) encontramos lo que vería el usuario nada más empezar la aplicación (en este caso, el usuario solo seré yo, desarrollador del sistema, ya que es una versión para pruebas). En ella podemos encontrar varios elementos informativos pero la gran mayoría no son nuestro. Como se ha comentado anteriormente, la interfaz se ha cogido de un ejemplo de Windows. Realmente lo que se deseaba era algo rápido para poder comenzar a probar la aplicación y, a la vez, poder comprobar los movimientos realizados. Los elementos marcados con con un número uno[1] serán informativos anteriores que no se quitaron. El único elemento reutilizable, por el cual hemos seleccionado la interfaz de pruebas, será la caja negra donde mostrará el esqueleto que reciba la Kinect[2]. En este estado no se ha reconocido a ningún usuario frente a la cámara. Aún así, el sistema se conecta y se mantiene en espera de un nuevo usuario.

Una vez reconoce a un usuario, se comenzará la ejecución. Se desarrolló solamente de gesto el cuadrado (nunca llegó a reconocerse) y movimientos cada noventa grados. En la **ilustración 4-2** se muestra cómo el sistema reconoce un movimiento de la mano a la izquierda (se reconocerá ya que se activa la imagen de la flecha izquierda[3]). Junto a la flecha izquierda, encontraremos las demás flechas y el cuadrado. Se mantendrán desactivados hasta que se ejecute el movimiento o el gesto. Además, se puede comprobar el recorrido seguido por la mano gracias a la línea de puntos azules que encontramos en forma de estela[4].

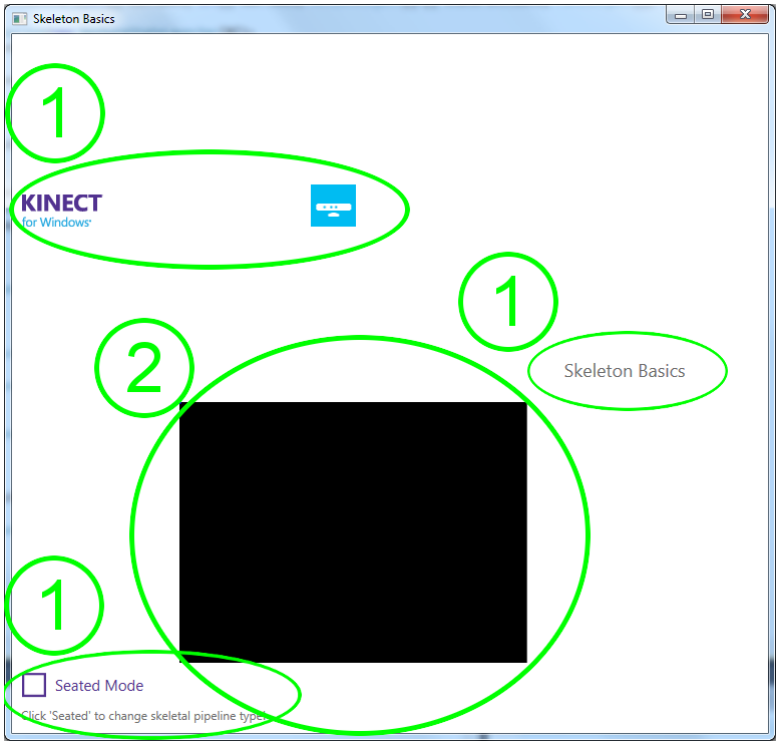


Ilustración 4.2: *Interfaz de prueba. Primera implementación. Sin usuario.*

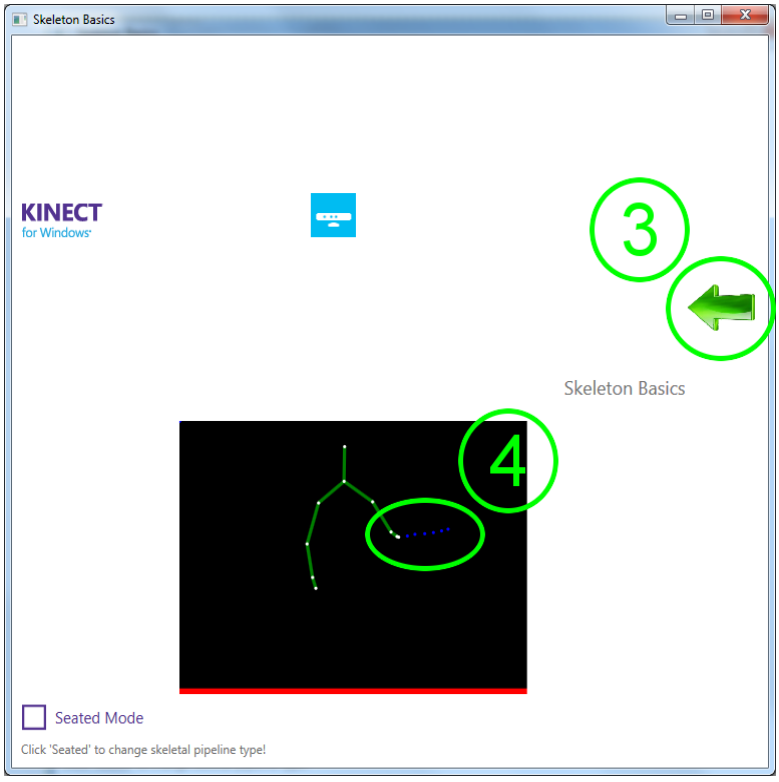


Ilustración 4.1: *Interfaz de prueba. Primera implementación. Izquierda.*

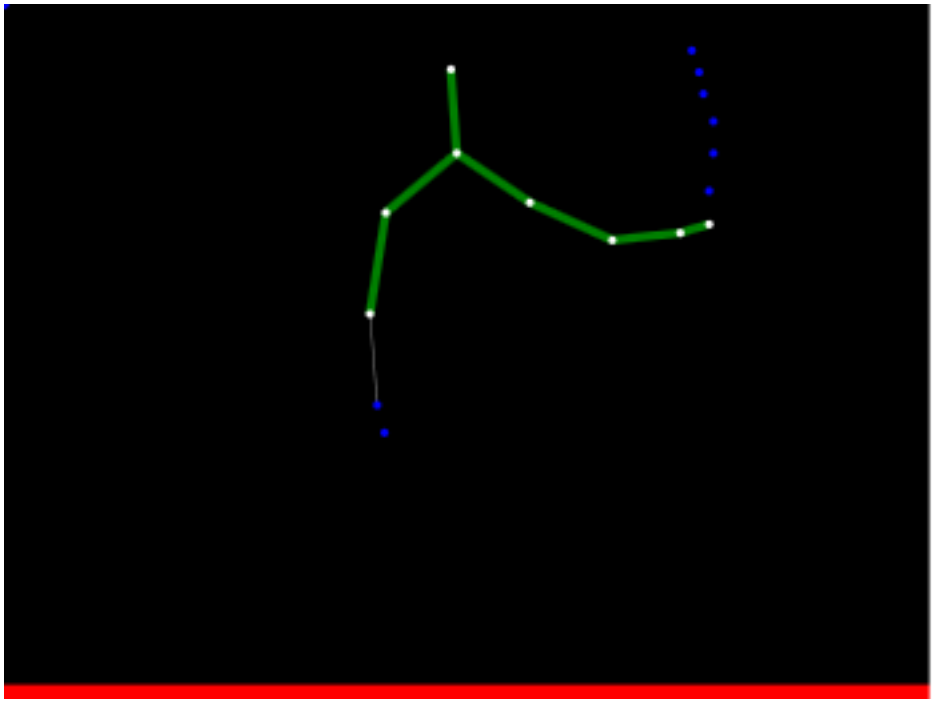


Ilustración 4.3: *Error en el análisis de puntos en la Kinect.*

Finalmente, como un pequeño detalle, se adjunta la **ilustración 4-3**. En esta se puede comprobar el problema que tiene la Kinect al devolver los puntos y el problema que hay al estimarlos. Como se puede comprobar, la línea que sigue el recorrido de la mano no es perfecta. Al contrario, ni hace una curva perfecta, ni hace una recta como el usuario supuestamente está haciendo (en el ejemplo se trataba de hacer una recta vertical hacia abajo).

4.1.2 Librería

La primera solución que se propuso fue la de crear un árbol de estados. Al comienzo de la aplicación se crearía el mismo con los posibles estados que se pueden tomar y dejaría un puntero al estado inicial. En este estado, se puede alcanzar cualquier otro estado o, incluso, mantenerse un tiempo indefinido sin moverse. Además, según el usuario avanza a un estado nuevo, el árbol vuelve a crear un nuevo puntero al estado inicial (con lo que conseguimos que se estén comprobando los gestos en todo momento, aunque el usuario haya hecho un movimiento indebido).

Cada estado mantendría una función propia (si así la definía el desarrollador). Esta función se lanzaría mientras el usuario tuviera un rastreador apuntándole. También se usará esa función en el caso de que el estado sea final: se lanzaría la acción que se deseara ejecutar con el gesto.

Se dividió en dos paquetes: el rastreador de estados (**StateTracker**) y el rastreador de gestos (**GestureTracker**). Este segundo paquete utilizaba los servicios del primero. Se diseñó de tal manera que el código fuera reutilizable. A continuación ,se describirá el contenido de cada uno, su funcionamiento y su implicación con el sistema.

4.1.2.1 StateTracker

Paquete encargado de la gestión del árbol de estados. Este árbol no será una máquina de estados convencional, se ha desarrollado de tal manera que incluya funcionalidades que se deseaban en el proyecto. Mantendrá un árbol de estados genérico. Cualquier desarrollador que necesite usarlo podrá crear sus propias clases extendiendo de las aportadas en este paquete. Los diferentes componentes del mismo son:

- **StatesTree.** Clase que implementa el árbol. Representará la estructura principal del árbol sirviendo como nodo principal.
- **State.** Implementa los estados del árbol. Contendrá los datos del estado y los lanzadores hacia los siguientes.
- **StateLauncher.** Lanzador del estado. Cada estado tendrá un número de lanzadores (puede ser cero en caso de ser un estado final). Los lanzadores serán los encargados de comprobar si se cumplen todos los requisitos para saltar a un nuevo estado. Contendrán el estado siguiente.
- **Tracker.** Puntero a un estado. Indicará en qué parte del árbol se están haciendo comprobaciones. Siempre habrá al menos un tarcker en el sistema, el del estado inicial.

4.1.2.2 GestureTracker

Extenderá de las clases definidas por el **StateTracker**. Dará al árbol de estados la funcionalidad del reconocimiento de los gestos. Cada estado será un movimiento atómico que se esté realizando. Cuando se cambie de movimiento será cuando el lanzador encuentre un cambio en la dirección de los puntos. Si encuentra un estado al que cambiar con ese nuevo movimiento, saltará. Si no, destruirá el rastreador.

El **GestureTracker** estará compuesto por:

- **GestureStatesTree.** Extiende el árbol genérico y le otorga la funcionalidad de reconocimiento de gestos.
- **GestureState.** Al igual que el **GestureStatesTree**, extiende un estado genérico para otorgarle las propiedades de los gestos.
- **GestureStateLauncher.** Define un lanzador que reconozca los gestos.
- **AtomicGesture.** Clase que implementa a un movimiento atómico. Define los campos necesarios para poder diferenciar unos gestos entre otros y la función para comprobar si ese movimiento es el deseado.
- **Gesture.** Gesto por excelencia. Componente que definirá los gestos que buscará el sistema. Estará definido por una serie de gestos atómicos (puede tener varias opciones para llevar a cabo un mismo gesto).

Para un ejemplo de definición de gestos, acudir al **Anexo A**.

4.2 Problemas con la primera implementación y decisión de cambio

Hubo una serie de problemas por los que me vi obligado a cambiar la implementación. A continuación detallaré los más graves y como se subsanaron.

- **Depuración.** En la primera implementación la depuración era muy difícil. Fue ya el problema que hizo cambiar la misma. La búsqueda y corrección de errores se hizo casi imposible. Las pruebas con la cámara eran inviables: la Kinect devuelve bastantes más de un punto por segundo. Si poníamos un punto de interrupción no se podía llevar a cabo el movimiento correctamente. Además, era muy difícil insertar los puntos a mano, tampoco podíamos hacer una depuración de lo que debería ser para comprobar el funcionamiento. Por ello se tomó la decisión de cambiarlo. La solución fue que al rastreador le pasaríamos el punto directamente. Así podríamos crear los puntos con el formato adecuado para ello. Además se hizo que cada gesto estuviera compuesto de un número determinado de movimientos. Con esto realmente solo se debía esperar aproximadamente medio segundo e introducir un punto, podíamos crear un banco de puntos de prueba para comprobar el funcionamiento.

- **Gestos.** La definición de un gesto, aunque se trató de hacer muy flexible para poder definir los gestos que se desearan, acabó siendo demasiado complicada. Definir un gesto implicaba programarlo, y cada nuevo gesto era un mundo. En la primera implementación cada gesto podía evaluar cuando se cambiaba de estado, qué se hacía en cada estado, cómo se salía del árbol si no se encontraba el final... Tenía muchas posibilidades, pero era demasiado complicado. En cambio, se decidió cambiar el sistema de tal manera que la librería no sería la encargada de lanzar ninguna acción. Ahora se le inserta un punto y si ese punto hace el movimiento final de un gesto, se le devuelve al desarrollador el identificador de dicho gesto para que lo trate por si mismo. Además el hecho de eliminar el árbol de estados y definir unos movimientos mucho más generales y amplios consiguió que la definición de los gestos fuera mucho más simple: simplemente se crearía un XML con los identificadores de los movimientos que contiene ese gesto en orden.
- **Superposición de gestos.** En la primera implementación el desarrollador del gesto indicaba cuándo se salía del árbol de manera positiva. Eso podía inducir a error y, por ejemplo, si deseabas llevar a cabo un cuadrado y en el banco de gestos tienes el cuadrado y un gesto para un barrido a la derecha, podía llevar a que se ejecutara el gesto a la derecha y el cuadrado, aunque el usuario solo deseara hacer el cuadrado. Para solucionar esto se tomó también la decisión de tener unos gestos rígidos. Una vez inicias el sistema, este sabe si tus gestos tienen uno, dos, cuatro o los movimientos que se deseen. El cuadrado ahora ocuparía cuatro movimientos y el barrido a la derecha también, por lo que no habría confusión. El problema de esto fue que se incrementó el tiempo de respuesta para gestos simples, pero se amplió la solidez del sistema.
- **Puntos extraños.** La Kinect no es un dispositivo totalmente fiable. Hay veces que el sistema devuelve puntos que no deberían ser. Otro de los beneficios de restringir los gestos a cuatro movimientos fue ese, ahora aunque un punto se salga de lo normal en medio, no pasa nada, no se evaluará, y en caso de ser el último, el error no será tan grande como para suponer demasiados problemas. La otra solución posible que se tuvo en cuenta fue el filtrar los puntos primero. Habría sido la mejor, pero el desarrollo de ese filtrado se salía de los límites del trabajo de fin de grado, por lo que se recomendará como trabajo futuro.

- **Tamaño del movimiento.** Cuando defines un movimiento en la primera implementación haces uso de un tamaño para el mismo. Has de definir el máximo y el mínimo. El problema llega cuando más de una persona usan el sistema: cada uno hará el gesto a su manera, con lo que restringir de esas maneras haría que algún usuario no consiguiera realizar algún gesto. Se decidió eliminar el tamaño del movimiento. Ahora los movimientos se definen con un vector del cual solo se tiene en cuenta el ángulo formado con respecto al suelo. El tamaño se incluye como propiedad, pero no lo hace obligatorio. Ahora se pueden definir dos gestos iguales tales como cuadrado pequeño y cuadrado grande, pero también se puede dejar abierto el sistema y que simplemente sea un gesto llamado cuadrado. Esto elimina problemas y, además, hace más sencillo la definición de movimientos o gestos.
- **Dependencia con la Kinect.** Aunque no fuera un error como tal, si que aportó puntos para cambiar la implementación. Antes todo el sistema estaba muy rígido con respecto a los puntos: no intervenía el desarrollador en ellos. Eso implicaba que si en un futuro se quisiera cambiar de dispositivo o, incluso, que cambiara la API de la Kinect, el sistema se quedaría obsoleto. Ahora se ha decidido dejar libre al rastreador de la Kinect, no realiza nada con ella. En cambio, se ha generado una clase que se encarga de todo lo referido al dispositivo. Ahora si se necesita cambiar el dispositivo solo habría que hacer la clase y transformar los puntos devueltos en los que reconoce el sistema, pero todo el reconocimiento de gestos y movimientos se mantendrá igual.

4.3 Segunda implementación

Tras el fallo inicial y rediseño del sistema se continua con el trabajo. En esta nueva implementación se decide llevar a cabo los cambios deseados con la anterior y, además, tratar de hacer más simple el sistema.

Como primera elección, se plantea dejar de programar los gestos y pasar a leerlos de algún tipo de fichero. Se toma la decisión de usar XML para ello. Es fácil de leer, usar y modificar (siempre y cuando no se extienda demasiado su contenido, algo que no pasará con nuestros objetos).

Además de ello, se comienza a usar un sistema de vectores para reconocer los gestos. Esta vez no tenemos un máximo y un mínimo, lo que se desarrolla es una manera de cambiar de vectores tridimensionales a vectores bidimensionales y se los trata como si en un plano se encontraran. A partir de ellos se obtiene los grados que se han realizado con respecto al eje de las *X*s y se usan esos grados. Nunca marcarán la amplitud del gesto, esto se mantiene abstraído en unas propiedades que se incluirán junto al gesto (además de otras propiedades como movimientos proporcionales).

Ahora los gestos se comprobarán cada vez que el sistema reciba un punto, lo que implica que será cada vez que trate de reconocer un movimiento. Al inicio del sistema se determinará un tiempo que se usará para medir ese movimiento. Cada vez que cumpla ese tiempo el sistema recibirá el punto, evaluará si hay un número suficiente de puntos como para obtener un gesto y, en caso de ser así, comprobará el gesto. Esto supone una reducción drástica de trabajo de procesador.

Por último, aunque solo válido durante el desarrollo, se crea un sistema de registros para los movimientos y gestos. Ya no se tratará de depurar la aplicación con el Microsoft Visual Studio si no que se comprobarán en los registros los puntos recibidos y tratados, los movimientos reconocidos y los gestos que se reconocen y/o pierden.

4.3.1 Interfaz

La aplicación finalmente se ha dividido, como se propuso durante el diseño, en dos subcomponentes: el **Calibrador** y la interfaz **General**. Todo se mostrará en una ventana de Windows, cambiando de página según necesidad (de un subcomponente se pasará a otro sin cambio entre ventanas). Se ha desarrollado todo en tonos de gris para dar un aspecto sencillo pero pulido a la interfaz. Todas las interacciones del usuario con la misma se harán a través o de botones o de la propia Kinect.

4.3.1.1 Calibrador

Nada más iniciar la aplicación (**Ilustración 4.4**) el usuario podrá elegir si desea ayuda para la calibración o calibrar directamente. Se ha suprimido el botón de iniciar sin calibrar, fallando en uno de los requisitos [**REQ (1)**], ya que el sistema, si le indicas el tiempo por defecto, no se ajusta nada bien al usuario. Se ha decidido que, al menos hasta que se desarrolle un sistema para guardar configuraciones, se deberá realizar siempre la misma.

Una vez presionado el botón de «Ayuda»[2] se nos mostrará una película indicándonos los pasos a seguir durante la calibración (**Ilustración 4.5**). Irán apareciendo cuadros de texto[3] con las instrucciones para la calibración [**REQ (2)**]. Se ha desarrollado el efecto como si de una máquina de escribir se tratase: las letras del texto aparecen progresivamente una a una[**REQ (5)**].

Cuando termine de escribirse el texto en la pantalla saltará directamente a la siguiente pantalla (**Ilustración 4.6**). En esta nueva sección se detallará el funcionamiento de los elementos multimedia que aparecerán durante la calibración [**REQ (3)**]. En primer lugar se explicará el círculo verde[4], que servirá para indicar al usuario cuándo ha sido reconocido por la Kinect (cuando no sea reconocido se mantendrá gris). Después, se explicará el círculo rojo[5] (**Ilustración 4.7**). Este, una vez seas reconocido por la Kinect (el circulo verde no desaparece), parpadeará en rojo. Cada vez que parpadee en rojo el usuario tendrá que, como explica la ayuda, realizar un cuadrado con la mano derecha.

Ilustración 4.5: *Interfaz. Segunda implementación. Ayuda (1).*

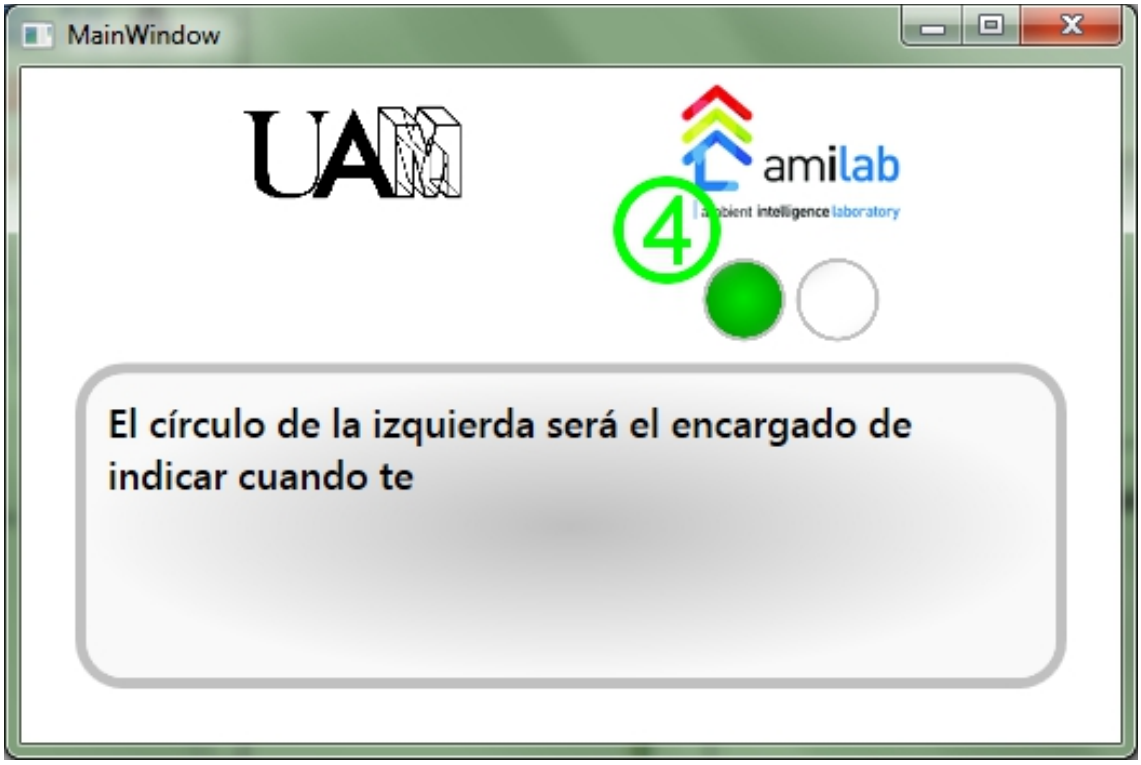


Ilustración 4.6: *Interfaz. Segunda implementación. Ayuda (2).*

Ilustración 4.7*: Interfaz. Segunda implementación. Ayuda (3).*

Cuando termine de explicar los elementos pasará a mostrar un sencillo ejemplo de uso (**Ilustración 4.8**). Aquí se detallarán los elementos restantes, la fila de cuadrados central[6], que serán la ayuda visual al usuario para tener siempre presente cuántos gestos ha reconocido la Kinect mientras está aplicando la actividad (si se pusiera un texto, debería ser muy grande, se ha optado por ello para ganar espacio y que sea mucho más visual e intuitiva). Cuando se terminen de mostrar los cinco cuadrados (**Ilustración 4.9**), será cuando el usuario haya llegado a calibrar bien, y así lo describen en la ayuda [**REQ (6)**].

Los últimos detalles se redactarán para el usuario (**Ilustración 4.10**) y se le devolverá a la página de inicio. En esta, si le ha quedado alguna duda, podrá volver a presionar la ayuda siempre que quiera [**REQ (7)**]. Con esto se finaliza la ayuda y se da paso a la calibración. Para ello, después de volver a ver la ayuda las veces que le sea necesario, el usuario presionaría el botón de calibrar[1]. Una vez presionado no se podrá volver atrás, a si que si el usuario deseara volver a ver la ayuda necesitaría cerrar la aplicación y volverla a abrir de nuevo para presentarse en la página de inicio.

Ilustración 4.8: *Interfaz. Segunda implementación. Ayuda (4).*

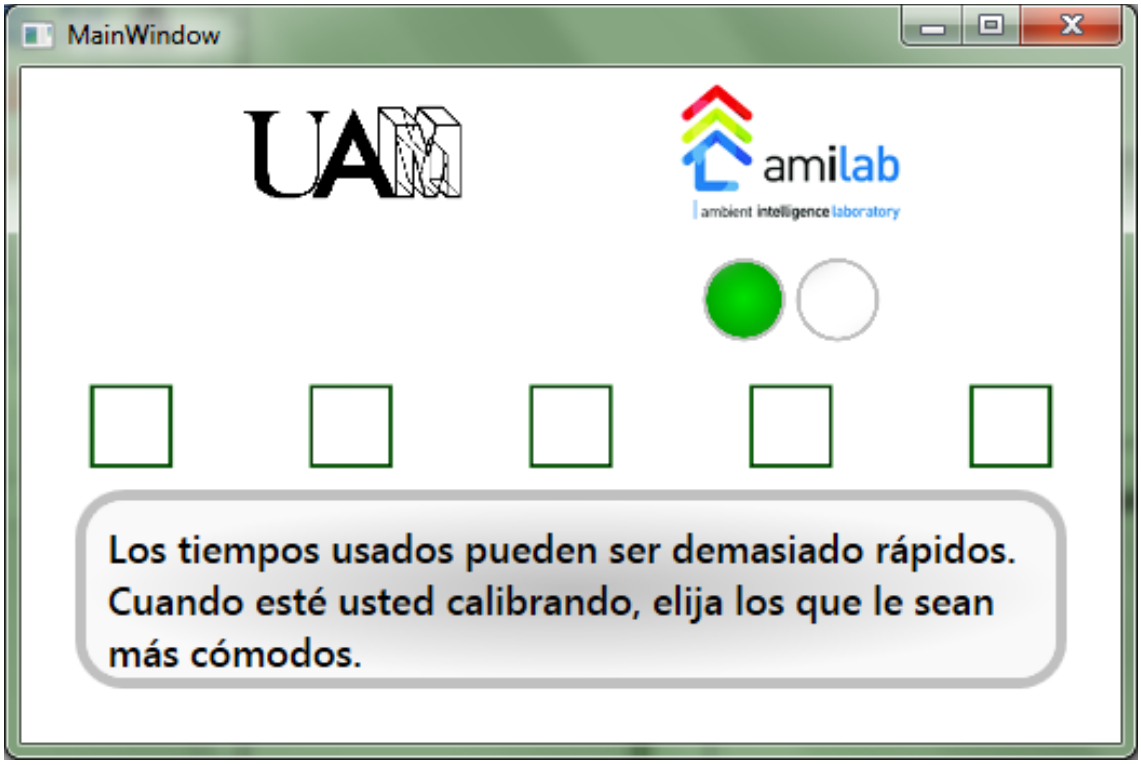


Ilustración 4.9: *Interfaz. Segunda implementación. Ayuda (5).*

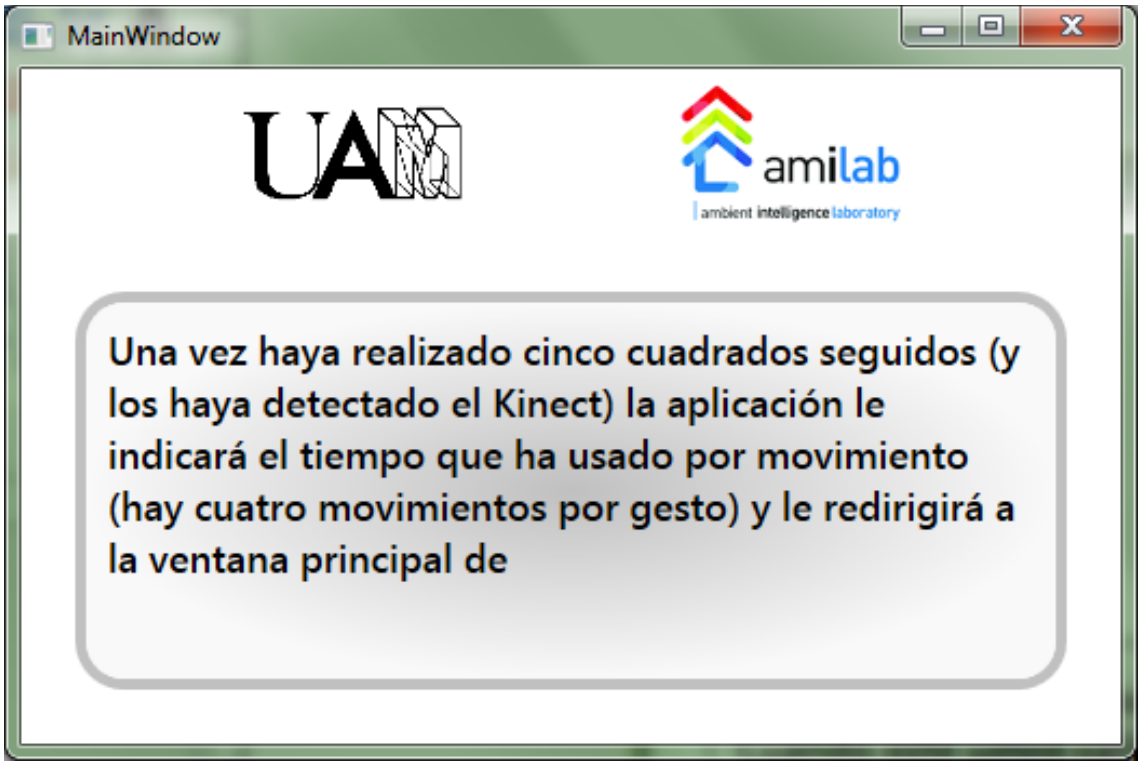


Ilustración 4.10*: Interfaz. Segunda implementación. Ayuda (6).*

En la página de calibración (**Ilustración 4.11**) se encontrarán los mismos elementos que se describieron en la ayuda [**REQ (4)**]. Se verá el indicador verde de usuario[4] [**REQ (9)**], el indicador rojo[5] de comienzo de gesto [**REQ (10)**], la tira de cuadrados[6] de estado [**REQ (11)**] y un cuadro de texto[7] para que mostrará el tiempo de movimiento en milisegundos [**REQ (14)**].

La calibración comenzará una vez el usuario se sitúe frente a la Kinect y sea reconocido. En ese momento, cada vez que se encienda el indicador rojo habrá que hacer un gesto. El sistema irá incrementando o decrementando el tiempo hasta encontrar uno que satisfaga las necesidades del usuario [**REQ (8)**]. Habrá que llevar a cabo cinco gestos para que finalice la calibración. Al final la calibración pudo no ser muy rápida [**REQ (13)**], pero realmente el tiempo obtenido se ajusta muy bien al que el usuario realiza. Una vez se ha calibrado (**Ilustración 4.12**), el indicador rojo se parará y se mantendrá gris y se activará un botón para acceder a la aplicación real [**REQ (15)**].

Se ha implementado sobre un color blanco para resaltar todos los elementos multimedia. Además, estos tienen unos colores bastante llamativos por lo que el usuario no tendrá problemas para diferenciarlos desde una distancia normal [**REQ (12)**].

Ilustración 4.11*: Interfaz. Segunda implementación. Calibración (1).*

Ilustración 4.12*: Interfaz. Segunda implementación. Calibración (2).*

4.3.1.2 General

Finalmente, llegamos a la aplicación final (**Ilustración 4.13**). Esta se dividirá en dos zonas [**REQ (19)**], una superior[9] con los gestos que se asocien a un elemento del entorno [**REQ (18)**] y otra inferior[10] con las acciones que se podrán llevar a cabo en esos elementos. Todos los gestos indicados en la *GUI* son reconocidos y ejecutados por el sistema [**REQ (16)**].

El funcionamiento de la herramienta será el siguiente:

- Primero el usuario comprueba los gestos y a qué elemento están asociados buscando el que desee activar o desactivar.
- Una vez encontrado el usuario realizará el primer gesto asociado a ese elemento. Si la Kinect lo reconoce se mostrará en pantalla resaltándolo[11] con un fondo verde [**REQ (20)**] (**Ilustración 4.14**).
- Ya resaltado, el usuario puede proceder al segundo gesto [**REQ (17)**]. Podrá realizar cualquier acción de las expuestas, por lo que solo tendrá que elegir entre las mismas y realizarla.
- Una vez realizado el segundo gesto, se activará o desactivará el actuador. En ese caso puede resultar que el usuario se haya equivocado y no haya seleccionado bien el elemento. Como el sistema mantiene el elemento activo hasta que se selecciona otro, solo tendrá que volver a ejecutar otro gesto secundario para devolver al sistema a su estado principal (si se había activado el elemento, ahora lo vuelve a desactivar) [**REQ (21)**].

Desgraciadamente, el sistema solo está preparado para soportar un usuario por sesión. En caso de desear cambiar de usuario se tendrá que apagar la aplicación y volver a iniciar para comenzar con un nuevo usuario.

A continuación se muestran udos capturas en las que se puede comprobar el funcionamiento del sistema. En la **ilustración 4.13** se muestra la ventana principal, sin ninguna acción por parte del usuario. Seguida de ella se muestra, **en la ilustración 4.14**, cómo quedará la GUI en cuanto el usuario seleccione un elemento.

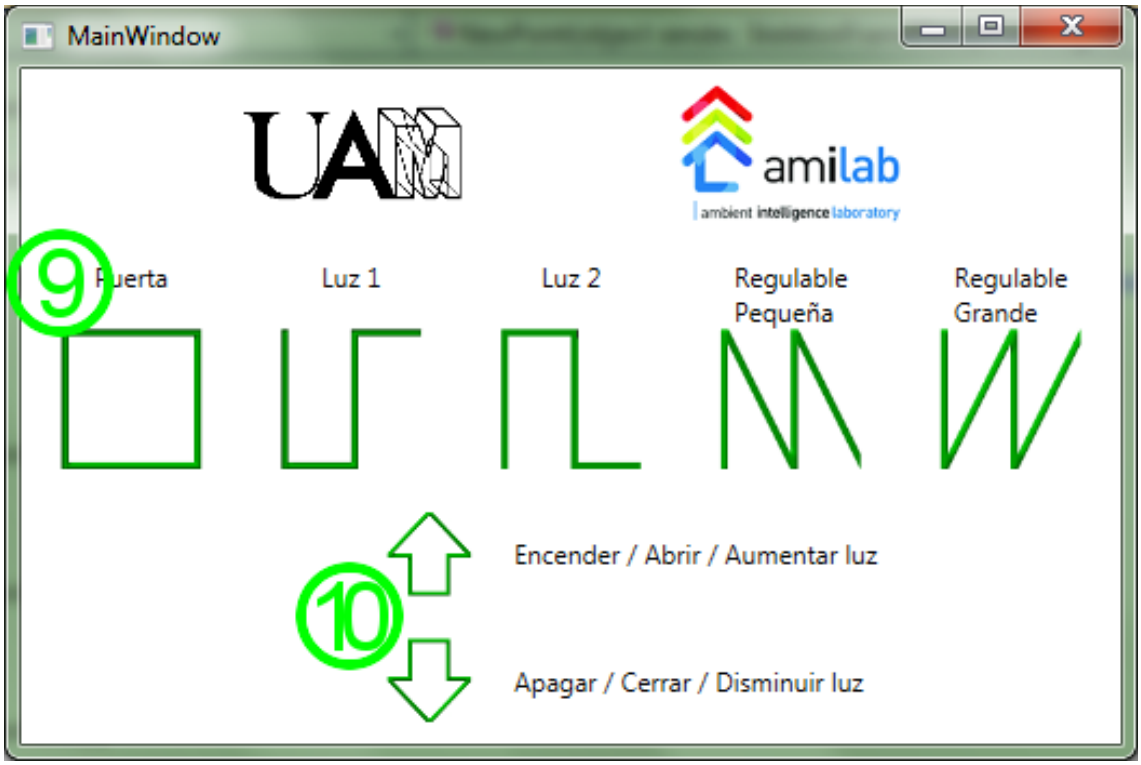


Ilustración 4.13: *Interfaz. Segunda implementación. Aplicación final (1).*

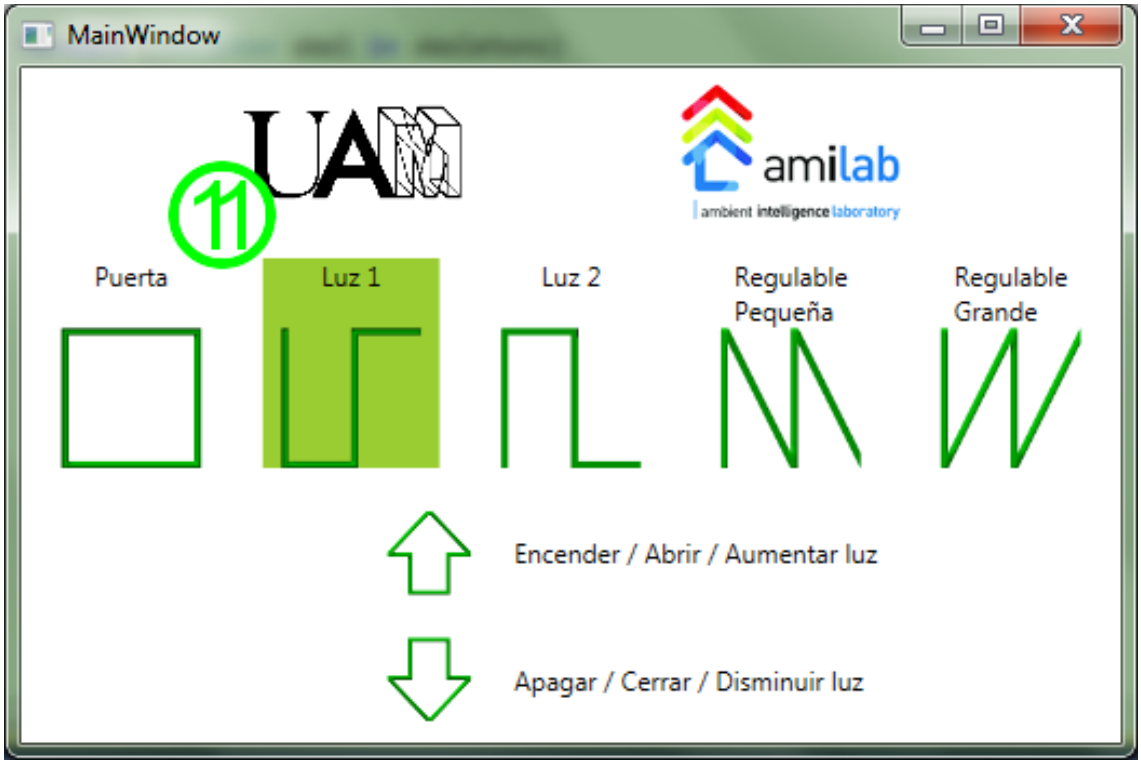


Ilustración 4.14: *Interfaz. Segunda implementación. Aplicación final (2).*

4.3.2 Librería

La solución final tomada ha sido a base de vectores. Se toman cinco puntos, realmente cuatro movimientos, que se evalúan para hallar el ángulo de los mismos con respecto al suelo. Además, se ha implementado la librería de tal manera que se abstraiga al usuario del uso del Kinect.

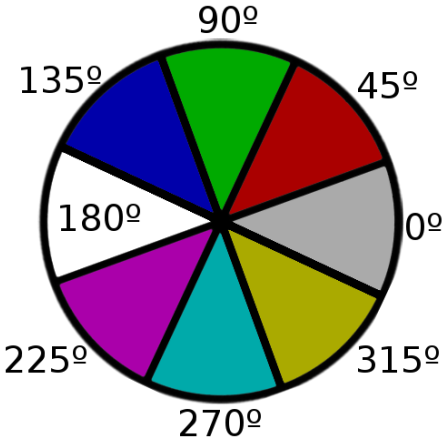
A continuación describiremos las funciones y comportamientos de los cuatro componentes más importantes de la librería:

- Gestos y movimientos. Definición de los mismos y tratamiento de los puntos.
- Módulo encargado del funcionamiento de la Kinect. Conectará la cámara, activará las funciones más necesarias de la misma, devolverá los puntos y parará todo.
- Módulo encargado del seguimiento de los puntos. Recogerá los puntos, los evaluará tal y como se explica un poco más adelante, comprobará los movimientos y los gestos y devolverá a la aplicación lo que haya encontrado.

4.3.2.1 Gestos

Aunque inicialmente se planteó llevar a cabo un banco de gestos fijos para la aplicación, finalmente se ha desarrollado un sistema para poder definir, mediante movimientos [**REQ (27)**], cualquier tipo de gesto.

Los movimientos vendrán definidos por vectores, que indicarán la dirección en la que se ha realizado el mismo. En la siguiente imagen, **ilustración 4.15**, se ven los que se han desarrollado para esta aplicación, aunque se podrían desarrollar más o, incluso, variarlos todos.



Además del vector, se tiene que asignar un margen de error, por ello se ha escogido tan reducido número de movimientos [**REQ (29)**]. La Kinect, como se comentó anteriormente, no genera un esqueleto totalmente preciso, además de que el ser humano realmente no es capaz de hacer los movimientos totalmente rectos y en la dirección indicada. Aún así, con todos los movimientos desarrollados (además del elemento vacío, cuando apenas se mueve), se podrían desarrollar 6561 gestos, un número lo suficiente algo para cualquier aplicación. Además, se ha dejado abierto el desarrollo para poder implementar gestos con un diferente número de movimientos.

En cualquier caso, los movimientos se han de asignar con un vector (el que indicará la dirección real) y un rango (por arriba y por abajo, realmente el movimiento tendrá un espacio de dos veces el rango definido).

Cada movimiento, como cada gesto, necesita de un número de identificación para poder distinguirse entre otros [**REQ (30)**]. Además, cada gesto sabrá los movimientos que posee gracias al identificador de los mismos, por lo que han de ser únicos.

La herramienta, cuando detecte un gesto, devolverá su identificador, con lo que también habrá que prestar atención a él [**REQ (31)**]. Además, se incluye otra restricción: no se puede usar el identificador *0*. Este se usará como «gesto vacío» [**REQ (34)**], gesto que, realmente, no es ningún gesto.

Los gestos serán algo más complejos que los movimientos. De primeras, poseerán un nombre, campo que se usará para describir el gesto. Además, contendrán una lista de movimientos que compondrán el gesto (que, como hemos comentado, serán relacionados con el identificador del movimiento). Por último, tendrán una lista opcional de propiedades propias de ese gesto [**REQ (32)**]. Se han desarrollado las siguientes propiedades:

- **numberRestrictedMovements.** Se incluirá dentro un número. Este será el número de movimientos tras el gesto que serán eliminados. Por ejemplo, hacemos un cuadrado con *numberRestrictedMovements* a uno. El siguiente movimiento no podrá detectar un gesto, está restringido, pero al segundo si.

- **ProportionalMovementDistance.** Incluirá dos movimientos y dos proporciones. Se indica la proporción de distancia que ha de tener un gesto respecto al otro.
- **MaximumDistance.** Distancia máxima de un movimiento. Incluir movimiento y valor.
- **MinimumDistance.** Igual que el *MaximumDistance* pero esta vez será mínima.

Todas las propiedades tendrán un valor llamado *err* que indicará un rango de error. Todas estas propiedades han sido desarrolladas pero no se han podido probar correctamente, con lo que se consideran como no realizadas.

Cómo se especificó, todos los gestos y movimientos estarán definidos en un fichero XML [**REQ (28)**] los cuales se situarán en la raíz de ejecución de la aplicación, dentro de la carpeta «Conf», y «Gestures» o «Movements» para gestos y movimientos respectivamente. Para una descripción más detallada de los movimientos y gestos, se incluyen los **Anexos B y C**.

En cuanto al tratamiento de los puntos, la Kinect devuelve un punto en tres dimensiones. Nuestra librería acepta puntos en dos dimensiones exclusivamente [**REQ (35)**], por lo que se ha tenido que desarrollar un sistema para convertirlos. En la **ilustración 4.16** se muestra el proceso seguido para pasar de una recta en tres dimensiones a dos.

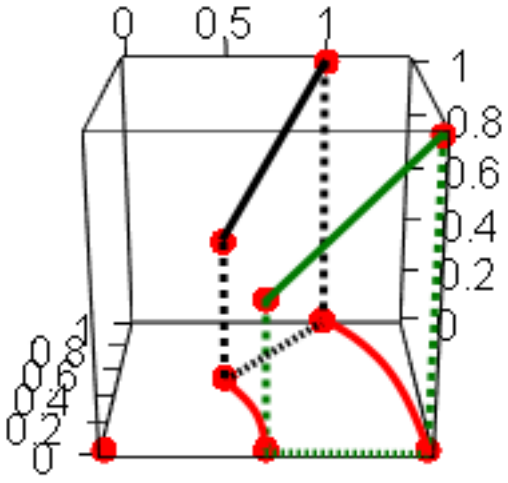


Ilustración 4.16: *Tratamiento de puntos.*

- En primer lugar, recogemos los dos puntos que formarán el gesto. Están en tres dimensiones, con lo que tendremos que tratarlos para conseguir las dos dimensiones. Ya que el eje *Z* es la profundidad, queremos traspasarlos al plano *Z=0*, olvidarnos de este parámetro, y centrarnos en los valores *X* e *Y* que consigamos.
- Proyectamos el movimiento en el plano *Y=0*, la coordenada Y será la misma, no necesitamos hacer ningún cambio sobre ella. En la ilustración se encuentra dibujado el proceso en negro.
- Cogemos el vector resultado y lo giramos para hacerlo encajar, con su tamaño, en la recta *Y=0*, *Z=0*. Pintamos este paso en rojo.
- Ahora solo nos queda unir las coordenadas *Y* a las *X* obtenidas. La recta verde será nuestro movimiento rotado.

Con nuestro vector ya en dos coordenadas podemos compararlo con los diferentes movimientos.

4.3.2.2 Runner

El *KinectRunner* es el encargado de la gestión del hardware. Sus funciones básicas son:

- Habilitar el flujo del esqueleto. Será la principal salida de información. Con el obtendremos los puntos necesarios para evaluar los gestos.
- Habilitar el flujo de profundidad. Se usará para la siguiente función.
- Devolver si hay algún usuario frente a la Kinect.
- Parar la Kinect cuando se finalice la aplicación.

Realmente el KinectRunner no tendrá muchas más funciones, pero su creación, como se comentó en el análisis, se mantiene ya que, gracias a ello, abstraemos al usuario del uso de la cámara. El único requisito del usuario será entregarle, en la creación, la función que se usará para encontrar cada punto. Además, para un trabajo futuro, se podría implementar diferentes *Runners* para el uso de la librería con otro tipo de hardware.

4.3.2.3 Tracker

Componente para tratar los puntos que devuelve el **Runner**. El *KinectTracker* (*rastreador*) iniciará el sistema para el reconocimiento de gestos. Será la clase que, básicamente, reciba un punto, compruebe si hay movimientos y, en caso afirmativo, si encajan con algún gesto de la lista.

Además, se han creado diferentes módulos para la gestión interna del sistema:

- **Movement.** Clase que guardará los datos de un movimiento. Posee el método *IsAchieved()* que devolverá verdadero en caso de que el vector suministrado a la función coincida con ese movimiento.
- **Gesture.** Clase que almacenará los datos de un gesto. Guardará los identificadores de los movimientos que contiene y la lista de propiedades que ha de cumplir. Tendrá funciones para recoger movimientos y comprobar las propiedades del mismo.
- **Property.** Reúne los atributos de una propiedad de un gesto. Toda propiedad creada por el desarrollador tendrá que extender de esta clase e implementar la función *PropertyAccomplished()* que indicará si esa propiedad se cumple para los puntos entregados.
- **XMLGestureReader.** Leerá y creará los movimientos y gestos. Tendrá varias funciones que devuelvan una lista de lo que se desee en cada caso.
- **Tracker.** Rastreador principal. Creará toda la estructura de gestos gracias a la clase anterior. Implementa la función *GestureTracked()* que devolverá el gesto encontrado. Además, se le ha creado la función *GestureTrackerCalibration()* la cual, en vez de devolver el gesto encontrado, devolverá el número de movimientos cumplidos.

Gracias a estas clases se consigue la gestión de los gestos. Realmente el funcionamiento, una vez estructurado correctamente, es realmente sencillo.

- Al **Tracker** le llega un punto. Si añadiendo ese punto consigue cinco (ya que son los necesarios para buscar cuatro movimientos) podrá comprobar los gestos. Comienza la búsqueda.

- Comprueba los dos primeros puntos, ya que serán los que indiquen el primer movimiento. Recorre la lista de movimientos posibles. Si estos dos puntos encajan con algún movimiento, se toma ese movimiento.
- Se recorre la lista de gestos. Si el primer movimiento de alguno de los gestos coincide con el encontrado anteriormente, se guarda el gesto. Si no, se elimina para siguientes comprobaciones.
- Se hacen las mismas comprobaciones pero en vez del primer y segundo punto con el segundo y tercero, que harán el segundo movimiento. Se van discriminando los gestos, cada vez se hace una lista más pequeña.
- Se sigue con los demás puntos. Si en algún momento la lista de gestos se vacía completamente se devolverá un gesto vacío. No se habrá encontrado ningún gesto.
- Si llegado al último punto queda un gesto, se devolverá su identificador.

Con esto la aplicación recibiría el resultado, mientras sigue recibiendo puntos y enviándolos. Es el momento de comprobar los gestos y, si se requiere alguna acción externa para cumplir con la acción asociada al gesto, se llamará a la **Demo**.

4.3.3 Demo

Último componente creado (ya que la **Pizarra** como se comentó en el capítulo de **Análisis y Diseño** ya se encontraba implementada en el laboratorio). Encargada de la conexión de la pizarra con nuestro sistema.

Las librerías para el uso de la pizarra están desarrolladas en Java. Para suplir la necesidad de crear unas nuevas librerías en C# y poder usarlas directamente desde nuestra aplicación, se ha creado un pequeño programa intermedio. Este se encargará de recibir un número y asociarlo con una acción. El entorno posee los siguientes elementos que podremos manejar: dos luces en el techo (lamp1 y lamp2), la puerta (puerta) y una lámpara (dimLamp1) con una luz pequeña y una grande regulable. A continuación detallo el código que se ha de enviar a la Demo y la acción que realizará:

- 0: Apagar lamp1.
- 1: Encender lamp1.

- 2: Apagar lamp2.
- 3: Encender lamp2.
- 4: Cerrar puerta.
- 5: Abrir puerta.
- 6: Decrementar en 10 puntos (de un máximo de 50 y un mínimo de 0) la dimLamp1 regulable.
- 7: Incrementar en 10 puntos la dimLamp1 regulable.
- 8: Apagar dimLamp1 pequeña.
- 9: Encender dimLamp1 pequeña.

La **Interfaz** y la **Demo** poseen los mismos códigos con lo que se ha decidido crear una comunicación por sockets (orientados a conexión) intermedia que, realmente, pasará el valor que desee la Interfaz hacia la Demo. Esta recogerá el valor y, con una simple comprobación, según el valor recibido enviará a la pizarra la acción a ejecutar.

No requiere casi uso del procesador, ya que el código es muy pequeño y simple, por lo que podrá encontrarse sin problemas en la misma máquina que nuestra aplicación. Aún así, si se desea, se podría arrancar sobre otra máquina sin problemas, con cualquier sistema operativo, ya que se ha desarrollado en Java.

Finalmente la única comprobación de que la acción enviada a la Pizarra se lleva a cabo es, básicamente, visual o auditiva. Habrá que comprobar el entorno, fijando nuestra atención en el elemento al que se ha enviado la acción.

CAPÍTULO 5 CONCLUSIONES Y TRABAJO FUTURO

Este trabajo de fin de grado, finalmente, ha sido bastante satisfactorio. Se han cumplido con casi todos los requisitos propuestos en el **capítulo 3** además de incluir ciertas mejoras con respecto a ellos. Se ha podido comprobar lo eficientes que son las NUIs. El ser humano las reconoce como más intuitivas y con este trabajo se ha visto que, aunque un sistema complejo venga detrás, el resultado puede ser tan simple como se desee y, por complicado que sea, si el funcionamiento es el correcto, se obtienen unos resultados muy satisfactorios para con el usuario.

Además, se ha conseguido indagar un poco en estos temas que a menudo parecen tan cercanos a la ciencia ficción que no se plantean normalmente. Las casas domóticas es algo que antiguamente no estaba a disposición de casi nadie, pero con esto se ha demostrado que, aunque siga teniendo que haber una inversión inicial, no es tan alta y resulta factible para una persona de clase media. Sigue siendo un pequeño lujo, un capricho, pero se han reducido costes para ser un capricho de muchas más personas.

Finalmente no se pudieron hacer grandes pruebas. Aún así, las pruebas de laboratorio que se llevaron a cabo indican que el sistema aún tiene que refinarse. El reconocimiento no es tan fiable como en otros casos, pero se podría subsanar con algunos cambios como los que se proponen:

- **Separar los puntos de los gestos.** Ahora mismo la librería recoge los puntos y los evalúa en un conjunto para encontrar los movimientos y el gesto. Se propone realizar un cambio en la estructura. Los gestos solo tendrán acceso a los movimientos, podrán acceder a una lista de los movimientos realizados, pero no a los puntos. En cambio, encontrar los movimientos se hará antes de ello. Con esto se conseguiría separar ambos módulos para poder implementar de una manera sencilla un análisis del movimiento en profundidad (que puede incluir algún tipo de filtrado para estimar, en caso de que el último punto no sea el correcto, cuál sería el movimiento correcto que deseaba el usuario).

- **Realizar un calibrador propio.** Aunque en nuestra librería se ha implementado un calibrador, no es del todo fiable y depende demasiado de la aplicación. Se podría realizar un calibrador especial que, aún a parte de la interfaz, pueda ser usado sin problemas. Además, se podría mejorar el mismo incluyendo más datos (la curvatura que hace cada persona al hacer un gesto ascendente, por ejemplo, se podría medir para calibrar mejor ese gesto y añadirlo al filtrado de puntos).
- **Configuración personalizada por usuario.** Ahora mismo la aplicación se ha de calibrar siempre al comienzo. Esto supone muchas veces una perdida de tiempo ya que los resultados no varían excesivamente. Se propone la creación de unos ficheros de configuración que, una vez iniciada la sesión por un usuario concreto, recoja los parámetros y active el sistema con los mismos. Además, se podría ampliar incluyendo un sistema de aprendizaje automático para que, durante el transcurso del sistema, este evalúe si has tenido algún cambio involuntario en tus movimientos (los realizas más lento, por ejemplo).
- **Crear un Runner para otro tipo de dispositivos.** La aplicación se ha desarrollado, como se comentó anteriormente, como un sistema autónomo del dispositivo que uses. Se podría llevar a cabo un Runner genérico, de donde extenderían los demás y, además, llevarlo a cabo con, por ejemplo, el WiiMote.
- **Poner en funcionamiento la aplicación con diferente número de movimientos por gesto.** Aunque se ha desarrollado pensando en esta funcionalidad, no está del todo definida. Habría que reparar algún error y desarrollar los movimientos y gestos con menos movimientos.
- **Preparar el sistema para utilizar más partes del cuerpo.** Una gran parte de la sociedad, por ejemplo, es zurda. Ahora mismo solo está preparado para encontrar gestos realizados con la mano derecha, aunque no sería complicado cambiar el parámetro para que reconociera de otros puntos o, incluso, de dos a la vez. Esto simplemente necesitaría de un mayor banco de gestos. Además, se podrían hacer gestos conjuntos con diferentes partes del cuerpo, con lo que el número de gestos se incrementaría mucho para sistemas complejos o que necesiten unos gestos más específicos.

Llevando a cabo la mayoría de lo propuesto se conseguiría un muy buen resultado y, si se deseara usar la librería en algún que otro proyecto, se podrían implementar cosas como:

- **En el ocio un mando para la televisión.** Con los gestos navegarías a lo largo de la interfaz de la misma. Además, se podría activar un número de movimientos por gesto diferente según en qué zona (dentro de un menú normalmente se necesitan movimientos muy básicos, necesitarían como mucho dos movimientos).
- **En el ocio también se podría realizar algún tipo de videojuego con el mismo.** Este podría orientarse al usuario en general, aunque también se podría extender a personas con necesidades especiales. Una pequeña ayuda para el aprendizaje de actividades diarias.
- **En el campo de la medicina, un sistema de control sin mandos.** En medicina hay multitud de situaciones en las que se necesita que todo esté esterilizado, limpio, a mano y rápidamente. Utilizando este sistema se podría llevar a cabo cualquier acción con una pantalla y la Kinect. No se necesitarían aparatos extra y, por ejemplo, el visionado de documentos, radiografías y otros ficheros multimedia se podría hacer directamente. Un sistema similar fue creado con el WiiMote, pero ya necesitabas un elemento intermedio que estuviera preparado para la situación.
- **Un sistema de ayuda para la rehabilitación, también en medicina.** Si incluyéramos en el sistema el seguimiento de diferentes partes del cuerpo se podría realizar una aplicación que ayudara a la rehabilitación pidiendo al usuario realizar diferentes tareas repetitivas e intentando que, a la vez, pareciera un juego.

Estos son unos pocos ejemplos de lo que se podría llegar a realizar. Es un sistema muy sencillo, con lo cual se podría adaptar a casi cualquier interfaz. Llegados a este punto solo habría que decidir qué hacer, y las necesidades que se encuentran en la actualidad.

REFERENCIAS

[1] AARTS, Emile; WICHERT, Reiner. "Ambient intelligence". *Technology Guide*. Berlin: Springer, 2009. pp. 244-249.

[2] ROBERTSON, Paul; LADDAGA, Robert; VAN KLEEK, Max. "Virtual mouuse vision based interface". *Proceedings of the 9th international conference on Intelligent user interfaces*. ACM, 2004. pp. 177-183.

[3] WEISER, Marc. "The world is not a desktop". *interactions*. ACM, 1994. pp. 7-8.

[4] STEINBERG, Gideon. "Natural User Interfaces". En *ACM SIGCHI Conference on Human Factors in Computing Systems*. 2012.

[5] *About the NUI group*. MOORE, Christian. Ref. 20/08/2013. Disponible en Internet: <http://nuigroup.com/log/about/>.

[6] EVERS-SENNE, Jan-Friso et al.. "Distributed Realtime Interaction and Visualization System". En *VMV*. 2002.

[7] BRANDL, Peter; RICHTER, Christoph; HALLER, Michael. "NiCEBook - Supporting Natural Note Taking". En *CHI 10: Proceedings of the eight annual SIGCHI conference on Human factors in computing systems*. 2010.

[8] MCARTHUR, Victoria; J. CASTELLUCCI, Steven; MACKENZIE, I. Scott. "An empirical comparison of "wiimote" gun attachments for pointing tasks". *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2009. pp. 203-208.

[9] SHIH, Ching-Hsiang et al.. "Assisting children with Attention Deficit Hyperactivity Disorder actively reduces limb hyperactive behavior with a Nintendo Wii Remote Controller through controlling environmental stimulation". *Research in Developmental Disabilities*. Elsevier, 2011. pp. 1631-1637.

[10] W. JOHN, Nigel. "Design and implementation of medical training simulators". *Virtual Reality*. Springer-Verlag, 2008. pp. 269-279.

[11] OIKONOMIDIS, Iason et al. "Tracking hand articulations using Kinect". En *BMVC*. 2011.

[12] FILIPE PEREIRA, Joel. *Autonomous parking using 3D perception*. 2012.

[13] CHANG, Yao-Jen; CHEN, Shu-Fang; HUANG, Jun-Da. "A Kinect-based system for physical rehabilitation: A pilot study for youngadults with motor disabilities". *Research in Developmental Disabilities*. Elseiver, 2011. pp. 2566-2570.

[14] MISTRY, Pranav; MAES, Pattie. "SixthSense: A Weareable Gestural Interface". En *ACM*. 2009.

[15] DAVE, Ankit et al. *Project Mudra: Personalization of Computers using Naturla Interface*. En International Journal of Computer Aplications. 2012, pp. 42-46.

[16] BELONGIE, Serge; MALIK, Jitendra; PUZICHA, Jan. "Shape Matching and Object Recognition Using Shape Contexts". En *IEE Transactions*. 2002.

[17] PLAGEMANN, Christian et al.. "Real-time Identification and Localization of Body Parts from Depth Images". En *IEE*. 2010.

[18] SHOTTON, Jaime et al. *Real-Time Human Pose Recognition in Parts from Single Depth Images*. En Communications of the ACM. 2013, pp. 116-124.

[19] GANAPATHI, Varun et al. "Real Time Motion Capture Using a Single Time-Of-Flight Camera". En *IEE*. 2010.

[20] BOBICK, Aaron F.; DAVIS, James W.. *The Recognition of Human Movement Using Temporal Templates*. En Pattern Analysis and Machine Intelligence. 2001, pp. 257-267.

[21] CHEN, Duan-Yu; SHIH, Sheng-Wen; LIAO, Hong-Yuan Mark. *Human Action Recognition Using 2-D Spatio-Temporal Templates*. En IEE International Conference on Multimedia and Expo. 2007, pp. 667-670.

[22] LI, Wanqing; ZHANG, Zhengyou; LIU, Zicheng. *Action Recognition Based on A Bag of 3D Points*. En Computer Vision and Pattern Recognition. 2010, pp. 9-14.

[23] SCHLÖMER, Thomas et al. "Gesture Recognition with a Wii Controller". En *ACM*. 2008.

[24] REN, Zhou et al. "Robust Hand Gesture Recognition with Kinect Sensor". *Proceedings of the 19th ACM international conference in Multimedia*. ACM, 2011. pp. 759-760.

[25] *Kinect para Xbox 360*. Windows. Ref. 20/08/2013. Disponible en Internet: <http://www.xbox.com/es-ES/Kinect>.

[26] *Componentes de Kinect*. Windows. Ref. 15/08/2013. Disponible en Internet: <http://support.xbox.com/es-ES/xbox-360/kinect/kinect-sensor-components>.

[27] *What is Wii?*. Nintendo. Ref. 15/08/2013. Disponible en Internet: <http://www.nintendo.com/wii/what-is-wii>.

[28] *PlayStation Move*. Sony. Ref. 15/08/2013. Disponible en Internet: <http://es.playstation.com/psmove/>.

[29] *PlayStation Eye*. Sony. Ref. 15/08/2013. Disponible en Internet: <http://es.playstation.com/ps3/peripherals/detail/item78899/PlayStation®Eye/>.

[30] *Visual Studio 2013 Preview.* Microsoft. Ref. 16/08/2013. Disponible en : <http://www.microsoft.com/visualstudio/esn/2013-preview>.

[31] *Welcome to NetBeans*. Oracle. Ref. 17/08/2013. Disponible en Internet: <https://netbeans.org>.

[32] *Eclipse*. The Eclipse Foundation. Ref. 20/08/2013. Disponible en Internet: <http://www.eclipse.org>.

[33] *AmiLab*. UAM. Ref. 20/08/2013. Disponible en Internet: <http://amilab.ii.uam.es>.

[34] *Download Kinect for Windows Developer Toolkit v1.6 from Official Microsoft Download Center.* Microsoft. Ref. 20/08/2013. Disponible en Internet: <http://www.microsoft.com/en-us/download/details.aspx?id=34807>.

GLOSARIO

AI – *Ambient Intelligence*

Inteligencia Ambiental. Integración de la informática en el entorno de la persona de forma que los ordenadores no se perciban como objetos diferenciados.

AMILAB – *Ambient Intelligence Laboratory*

Grupo de investigadores de la Escuela Politécnica Superior de la Universidad Autónoma de Madrid interesados en la aplicación de Inteligencia Ambiental en Entornos Activos.

API – *Application Programming Interface*

Interfaz de programación de aplicaciones. Es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

CLI – *Command-Line Interface*

Método que permite a las personas dar instrucciones a algún programa informático por medio de una línea de texto simple.

GPU – *Graphics Processing Unit*

Unidad de procesamiento gráfico. Dedicado al procesamiento de gráficos u operaciones de coma flotante para aligerar la carga de trabajo del procesador central en aplicaciones como videojuegos o 3D interactivas.

GUI – *Graphical User Interface*

Interfaz gráfica de usuario. Utiliza un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz.

IDE – *Integrated Development Environment*

Entorno de Desarrollo Integrado. Programa informático compuesto por un conjunto de herramientas de programación.

NUI – *Natural User Interface*

Interfaz natural de usuario. Es aquella en la que se interacciona con un sistema, aplicación, etc. sin utilizar sistemas de mando o dispositivos de entrada de las GUI.

OOP – *Object-Oriented Programming*

Programación orientada a objetos. Paradigma de programación que usa los objetos en sus interacciones para diseñar aplicaciones y programas informáticos.

ANEXO A. CREACIÓN DE UN GESTO. PRIMERA IMPLEMENTACIÓN

A.1 Definición de movimientos

El primer paso a llevar a cabo será la definición de movimientos. Habrá que comprobar qué movimientos va a poder reconocer el sistema. Para el ejemplo definiremos dos movimientos: vertical hacia arriba y vertical hacia abajo.

A.1.1 Vertical hacia arriba

Todos los movimientos deben extender de la clase **AtomicGesture**. Por ello, el primer paso es crear una nuevo objeto que incluya la librería que se ha desarrollado y una clase que extienda del AtomicGesture.

En segundo lugar habrá que definir su constructor y, además, hacer la llamada al constructor de la clase superior donde introduciremos los datos de los puntos. El primer punto indicará la dirección mínima y su velocidad y el segundo la dirección máxima y su velocidad. Tener en cuenta que las coordenadas verticales están invertidas. En nuestro caso se requiere capturar un movimiento hacia arriba, los puntos que hemos elegido son: como mínimo X=-15, Y=-40 y como máximo X=15, Y=-10.

El resultado del movimiento será:

```
class AtomicGestureUp : AtomicGesture
{
    public AtomicGestureUp()
        : base(new Point(-15, -40), new Point(15, -10))
    {
    }
}
```

A.1.2 Vertical hacia abajo

Seguiremos los mismos pasos del apartado anterior. En este caso los puntos elegidos serán: como mínimo X=-15, Y=10 y como máximo X=15, Y=40.

```
class AtomicGestureDown : AtomicGesture
{
    public AtomicGestureDown()
        : base(new Point(-15, 10), new Point(15, 40))
    {
    }
}
```

A.2 Definición de gestos

Para el ejemplo se tomará como gesto unicamente uno, *vertical*. Este contendrá tanto hacer un gesto vertical hacia arriba o un gesto vertical hacia abajo.

Al igual que para los movimientos, se incluirá la librería en el archivo. Además, se creará una clase que extienda de **Gesture**. Habrá que insertar en la variable **atomicGestures** todas las posibilidades del gesto (en nuestro caso solo dos, arriba o abajo). Se creará un array con los movimientos que tiene cada uno de esos gestos y se le incluirá a la variable.

También será obligatorio crear un constructor que llame al constructor superior enviándole las diferentes opciones que tiene el gesto (en nuestro caso dos), el número de gestos atómicos por función (en este caso solo uno) y la variable atomicGestures para que pueda guardarlos.

Por último, habrá dos métodos que serán de necesaria implementación. **CanChangeState** indicará si se puede pasar al siguiente gesto atómico o no. Se podrá añadir toda la lógica que el desarrollador desee (en nuestro caso comprueba simplemente que la nueva dirección sea una dirección permitida). **CreateStateLaunchers** será la usada para rellenar el árbol de estados cuando se inserte este gesto. Se crearán los lanzadores personalizados para los gestos.

A.3 Código final del gesto

```
  class Vertical : Gesture
  {
    private static AtomicGesture[] Up = {new AtomicGestureUp()};
    private static AtomicGesture[] Down = {new AtomicGestureDown()};

    private static AtomicGesture[][] atomicGestures = {Up, Down};

    public Vertical()
        : base(2, 1, atomicGestures)
    {}

    public Boolean True(Object[] data)
    {
      return true;
    }

    override public StateLauncher[] CreateStateLaunchers(int[] minTimes, int[] maxTimes,
Action<Object[]>[] stateFunctions, Func<Object[], Boolean>[] canStayFunctions)
    {
      ArrayList auxStateLaunchers = new ArrayList(new GestureStateLauncher[0]);
      int numberOfAtomicGestures = GetNumberOfAtomicGestures();

      for (int i = 0; i < GetDifferentGestureOptions(); i++)
      {
        GestureStateLauncher launcher = null;

        for (int j = 0; j < numberOfAtomicGestures; j++)
        {
          Boolean isFinal  = true;
          GestureStateLauncher[] launchers = new GestureStateLauncher[0];
          Action<Object[]> stateFunction = stateFunctions[numberOfAtomicGestures - (j + 1)];
          Func<Object[], Boolean> canStayInFunction =
canStayFunctions[numberOfAtomicGestures - (j + 1)];
          if (j != 0)
          {
            isFinal = false;
            launchers = new GestureStateLauncher[1];
            launchers[0] = launcher;
          }

          GestureState state = new GestureState(launchers, stateFunction, canStayInFunction,
isFinal);

          launcher = new GestureStateLauncher(state, minTimes[j], maxTimes[j],
numberOfAtomicGestures - (j + 1), CanChangeState);

          isFinal = false;
          launchers = new GestureStateLauncher[1];
          launchers[0] = launcher;
          stateFunction = null;
          canStayInFunction = True;
          state = new GestureState(launchers, stateFunction, canStayInFunction, isFinal);
```

```
          launcher = new GestureStateLauncher(state, 0, 1, numberOfAtomicGestures - (j + 1),
CanChangeState);
        }

        auxStateLaunchers.Add(launcher);
      }

      return (StateLauncher[])auxStateLaunchers.ToArray(typeof(StateLauncher));
    }

    public Boolean CanChangeState(Object[] objects)
    {
      int state = (int)objects[0];
      int option = (int)objects[1];
      Point dir = (Point)objects[2];

      if (atomicGestures[option][state].IsThis(dir))
        return true;
      return false;
    }
  }
```

ANEXO B. CREACIÓN DE UN MOVIMIENTO

B.1 Descripción del movimiento

Lo primero que hay que plantearse es: ¿cuál es la dirección del gesto que queremos crear? Una vez decidido, hay que crear el vector que indica esa dirección. Para ello se usa simplemente la ecuación del ángulo entre dos vectores.

$$\cos\alpha = \frac{u_1 \cdot v_1 + u_2 \cdot v_2}{\sqrt{u_1^2 + u_2^2} \cdot \sqrt{v_1^2 + v_2^2}}$$

El vector *u* será el vector que buscamos. El vector *v*, como es el vector sobre el que tomaremos el ángulo, podemos usar *(1, 0)*. Con esto la ecuación nos quedaría de esta manera:

$$\cos\alpha = \frac{u_1}{\sqrt{u_1^2 + u_2^2}}$$

Despejando la ecuación y tomando un valor de la *u* fijo (v. gr. u_1 lo igualamos a 1) encontraremos el vector deseado.

Nosotros, por ejemplo, usaremos uno muy sencillo, el vector de 0º. Lo cual indicaría un movimiento a la derecha. Si llevas a cabo toda la ecuación (aunque realmente no hace falta, es un vector que se puede hallar a ojo) obtenemos como resultado *u=(1, 0)*.

Por último solo habría que decidir el identificador del mismo y un error. Para el identificador podemos seguir un orden ascendente desde 0 (el cual será el nuestro para el ejemplo ya que es el primer movimiento creado). Para el error, el cual se encuentra en radianes, solo tendremos que poner el ángulo de diferencia hacia arriba o hacia abajo que queremos (en nuestro caso 0.35 ya que distribuiremos la gran mayoría de los ángulos y 0,35 radianes equivale a poco más de 20º. No queremos que haya un agujero demasiado grande sin evaluar).

B.2 Creación del archivo

El formato del XML es muy simple. Aquí se presenta el XML final del movimiento.

```
<?xml version="1.0" encoding="utf-8" ?>
<movimiento>
	<id>0</id>
	<Xpunto>1</Xpunto>
	<Ypunto>0</Ypunto>
	<error>0.35</error>
</movimiento>
```

ANEXO C. CREACIÓN DE UN GESTO

C.1 Descripción del gesto

Al igual que con los movimientos, lo primero a hacer es decidir qué gesto se va a realizar. Para este ejemplo se usará un cuadrado.

Una vez decidido, se elegirán los movimientos que comportan ese gesto. En el caso del cuadrado, por ejemplo, serían un movimiento a la derecha, uno arriba, uno a la izquierda y el último abajo. Con ellos, miraremos sus identificadores. En nuestro caso serán el 0 para la derecha, el 1 para arriba, el 2 para la izquierda y el 3 para abajo.

Por último, para finalizar con los atributos obligatorios, se decidirán un identificador y un nombre. El identificador será un número igual a 1 o superior, por lo que elegiremos el 1. Como nombre usaremos algo descriptivo del gesto, «Cuadrado».

Además, aunque como se comenta en el contenido del trabajo no se ha probado su correcto funcionamiento, podemos definir propiedades para el gesto. Un cuadrado ha de tener todos los lados iguales, incluiremos esa propiedad. Además, si se encuentra un cuadrado quiere decir que los movimientos de dentro son suyos, no de otra figura, por lo que se restringirán los 3 siguientes gestos.

C.2 Creación del archivo

Una vez definido todo el gesto, se escribirá el archivo XML y se guardará en la carpeta correspondiente. El código resultado del nuestro será el siguiente:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<gesto>
	<id>1</id>
	<nombre>Cuadrado</nombre>
	<movimientos>
		<m1>0</m1>
		<m2>2</m2>
		<m3>4</m3>
		<m4>6</m4>
	</movimientos>
	<propiedades>
		<numberRestrictedMovements>3</numberRestrictedMovements>
		<proportionalMovementDistance>
			<prop1>1</prop1>
			<prop2>1</prop2>
			<mov1>1</mov1>
			<mov2>2</mov2>
			<err>0.5</err>
		</proportionalMovementDistance>
		<proportionalMovementDistance>
			<prop1>1</prop1>
			<prop2>1</prop2>
			<mov1>2</mov1>
			<mov2>3</mov2>
			<err>0.5</err>
		</proportionalMovementDistance>
		<proportionalMovementDistance>
			<prop1>1</prop1>
			<prop2>1</prop2>
			<mov1>3</mov1>
			<mov2>4</mov2>
			<err>0.5</err>
		</proportionalMovementDistance>
		<proportionalMovementDistance>
			<prop1>1</prop1>
			<prop2>1</prop2>
			<mov1>4</mov1>
			<mov2>1</mov2>
			<err>0.5</err>
		</proportionalMovementDistance>
	</propiedades>
</gesto>
```